普通高等教育“计算机类专业”规划教材

操作系统教程

（Linux版）

毛玉萃　牛玉军　赵宏伟 编著

清华大学出版社

普通高等教育“计算机类专业”规划教材

操作系统教程（Linux 版）

毛玉萃　牛玉军　赵宏伟　编著

清 华 大 学 出 版 社

北　　京

内容提要

本书以操作系统的基本功能(处理机管理、存储管理、文件系统、设备管理和用户接口)为主线介绍操作系统的相关概念、基本原理和基本方法,对进程管理的相关问题:进程的概念、描述、状态机器转换、进程控制、互斥、同步、通信和死锁做了详细阐述。简单介绍了其他几种类型的操作系统,并对 Linux 操作系统进行了剖析。本书注重理论与实践相结合,每章都配有相关习题,最后一章安排了 6 个实验。

本书可以作为普通高等院校计算机科学与技术及相关本科专业的教学用书或参考书,也可作为计算机及相关专业考研的参考书,还可供计算机技术领域相关人员参考。

图书在版编目(CIP)数据

操作系统教程:Linux 版/毛玉萃,牛玉军,赵宏伟编著. --北京:清华大学出版社,2013
普通高等教育"计算机类专业"规划教材
ISBN 978-7-302-32376-1

Ⅰ. ①操…　Ⅱ. ①毛…　②牛…　③赵…　Ⅲ. ①Linux 操作系统－高等学校－教材　Ⅳ. ①TP316.89

中国版本图书馆 CIP 数据核字(2013)第 093530 号

责任编辑:白立军　战晓雷
封面设计:常雪影
责任校对:李建庄
责任印制:杨　艳

出版发行:清华大学出版社
网　　址:http://www.tup.com.cn, http://www.wqbook.com
地　　址:北京清华大学学研大厦 A 座　　**邮　　编:**100084
社 总 机:010-62770175　　**邮　　购:**010-62786544
投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈:010-62772015, zhiliang@tup.tsinghua.edu.cn
课件下载:http://www.tup.com.cn,010-62795954
印 装 者:北京嘉实印刷有限公司
经　　销:全国新华书店
开　　本:185mm×260mm　　**印　　张:**20.75　　**字　　数:**519 千字
版　　次:2013 年 9 月第 1 版　　**印　　次:**2013 年 9 月第1 次印刷
印　　数:1～2000
定　　价:35.00 元

产品编号:052401-01

前言

操作系统是现代计算机系统中最基本、最重要的系统软件。它是最接近硬件的第一层软件，是硬件功能的首次扩充；计算机系统中的任何其他软件均运行在操作系统所构筑的软件平台上。如果没有操作系统，人们使用计算机和利用网络进行各种活动将变得十分困难，甚至是不可能的。操作系统的主要功能是管理和控制计算机系统的各种硬件和软件资源。操作系统使计算机的各部件能够高效、协调地运行，以提高资源的利用率，同时为用户使用计算机系统提供一种方便、简单、直观的手段(即用户接口)。随着计算机体系结构的发展以及用户需求的不断变化，对操作系统的要求越来越高，其功能越来越强，实现难度也越来越大，所以操作系统一直是计算机科学与技术领域的重要研究对象之一。

操作系统在计算机系统的高效运行和管理过程中具有相当重要的作用，同时，用户或程序开发人员要对操作系统具有一定的认识和了解才能充分利用计算机系统的各种软、硬件资源，更好地实现所要完成的功能，所以学习和掌握现代计算机操作系统的基本原理和实现技术是十分必要的。

为了方便读者对操作系统的原理和实现技术有一个全面、系统的认识和理解，本书以操作系统的主要功能为主线，以其理论和实现技术相结合的方式进行组织；理论的论述力求简明、通俗、精练；实现技术以 Linux 操作系统为实例，对理论上难以理解的概念进行解释，力求形象、直观；并结合作者多年来在操作系统方面的研究以及教学过程中对相关理论、技术问题的理解和认识，全面地论述了操作系统的原理及其实现技术，并精选了典型问题作为案例加以讲解。同时，利用 Linux 操作系统源码开放等特点，广大科技人员和学生可以非常方便地上机认证，以获得更为直观的感性认识。本书除了介绍成熟的理论和技术之外，还对操作系统领域中的前瞻性问题和热点问题(例如多媒体、对称多处理、分布式系统、集群和安全操作系统等)进行了阐述，论述了这些操作系统的特殊之处和技术难点，力求起到抛砖引玉的作用。每章均配有类型丰富的大量习题，可以进一步提高对相关概念、理论和技术的理解。

本书撰写过程中，结合了作者 10 余年的教学经验，以及中等水平的普通高等学校计算机及相关专业的教学特点和学生特点。

近几年，操作系统作为计算机科学与技术以及相关专业考研的四门必考专业课程之一，使得操作系统这门课更得到了普遍重视。因此本书的内容还结合了这几年的考研大纲和试题。

本书内容的安排与组织情况如下：第 1 章为操作系统概述，主要介绍操作系统的基本概念、分类、基本特征、基本功能、组成、发展以及研究操作系统的观点，并对几种典型的操作

系统进行简单介绍。第 2 章为用户接口，除了作业的相关概念之外，还介绍操作系统提供给用户的 3 种接口形式。第 3 章为进程管理，主要介绍进程的概念、特性以及与程序、作业的区别，还介绍进程的描述、进程的状态和转换以及线程的概念。第 4 章为处理机管理，主要介绍计算机系统的分级调度及其算法，并对各种调度算法进行详细论述。第 5 章为存储管理，对内部存储器的管理问题进行讨论，主要包括各种管理方法如何实现内存的分配与回收、地址转换、共享与保护以及扩充等问题，并对内存的各种管理技术进行比较。第 6 章为文件管理，主要介绍文件管理中的基本概念、文件的逻辑组织和物理组织、文件存储空间的管理、磁盘的容错技术以及文件的共享、保护和保密等问题。第 7 章为设备管理，主要介绍设备管理的功能、作为主要外部存储设备——磁盘的驱动调度、与设备管理密切相关的技术（中断、通道和缓冲）以及设备的分配、虚拟设备等问题。第 8 章为进程的互斥、同步、通信和死锁，主要介绍进程的互斥、进程间的同步与通信以及死锁，并对有关进程通信的经典问题进行讨论。第 9 章为其他几种操作系统简介，主要介绍安全和安全操作系统、并行计算机操作系统、网络操作系统、分布式操作系统以及多媒体操作系统，主要介绍这几种操作系统的主要功能、特点和技术难点。第 10 章为操作系统实验，提供 6 个实验——编程接口实验，进程的创建、执行、终止实验，作业（进程）调度实验，动态页式存储管理实验，文件系统实验，以及进程互斥、同步、通信实验。

本书由毛玉萃、牛玉军和赵宏伟共同编写，其中第 2 章由赵宏伟编写，第 7 章由牛玉军编写，其余各章由毛玉萃编写，最后由毛玉萃进行了全书的统稿。

本书覆盖面广、内容丰富、技术性强、可读性好，可以作为广大计算机科学与技术工作者从事相关专业研究的参考书，也可以作为普通高等院校计算机科学与技术本科专业的教学用书或参考书，同时也可作为计算机及相关专业考研的参考书。

由于作者水平有限，书中一定存在不妥之处，敬请广大读者批评指正。

编著者

2013 年 7 月

目录

第 1 章　操作系统概述

1.1　操作系统的基本概念

众所周知，计算机系统由硬件（hardware）和软件（software）两部分组成。硬件包括CPU、存储器和输入输出设备等，是用户直接可见的部分。而软件是存储、运行在存储器、CPU 中的程序，是用户直接观察不到的部分，具有一定的抽象性，难以理解。软件一般分为系统软件、支撑软件和应用软件。应用软件是为了完成某种特定应用功能的专用程序，如 Office 办公软件，它运行在系统软件或系统软件和支撑软件所构筑的软件平台之上。支撑软件运行于系统软件之上，为应用软件提供开发环境和手段，以方便于应用软件系统的开发，如各种集成软件开发环境和各种中间件（middleware）等。系统软件是用于对计算机的软硬件资源进行管理并为应用程序提供服务的程序集合，如编译程序和解释程序等；操作系统（operating system）是最基本的系统软件，是最接近硬件的第一层软件，它负责管理计算机系统的各种软硬件资源，并为其他软件的运行提供支撑。

计算机系统中的硬件常被称为裸机（bare machine），它通常由电子、磁、机械、光学等部件所组成。按照冯·诺依曼结构，计算机系统可划分为 5 个组成部分：运算器、控制器、存储器、输入设备和输出设备。如果没有操作系统等系统软件的支撑，程序员只能采用机器语言来编写程序；这种使用二进制代码编程非常烦琐，不直观，可读性和移植性差；而且还需要程序员考虑程序运行过程中计算机的各个组成部分如何工作的具体细节，这对于程序员来讲要求很高，负担很重，难以充分发挥计算机硬件的效率。针对这种情况，人们在计算机硬件的基础上增加一层软件来自动管理计算机系统的软、硬件资源和控制程序的运行，将硬件的复杂性同程序员分离开来，并为用户提供使用计算机系统资源的简便手段，使用户能够高效、方便地使用计算机；通过这层软件，用户仅需将要计算机所做的工作用直观、简单、形象的语言（如高级语言）编好程序并提交给计算机，而程序的运行以及在其运行过程中所涉及的系统资源的分配和使用等完全由该层软件来做，这样，用户使用计算机就显得相当方便，而且轻松自如。人们把位于裸机上面的这层系统软件称为操作系统；由计算机硬件和操作系统所组成的计算机系统称为虚拟机（virtual machine），它具有比裸机更强的功能和更好的易用性，此时计算机系统的组成如图 1.1 所示。

<table>
<tr><td colspan="4">用户</td></tr>
<tr><td rowspan="5">计算机</td><td rowspan="4">软件</td><td colspan="2">应用软件</td></tr>
<tr><td colspan="2">支撑软件</td></tr>
<tr><td colspan="2">其他系统软件</td></tr>
<tr><td>操作系统</td><td rowspan="2">虚拟机</td></tr>
<tr><td>硬件</td><td>运算器、控制器、存储器、I/O控制器</td></tr>
</table>

图 1.1　计算机系统的组成

由上面的分析可知,操作系统是一种系统软件,是由若干程序所组成的集合,它负责计算机系统的全部软、硬件资源的分配、调度和管理,使系统高效、安全地运行,并为用户提供简单、直观、灵活的接口,以便用户使用计算机系统。

1.2　操作系统的基本功能

如上所述,操作系统负责分配、调度和管理计算机系统的全部软、硬件资源,并为用户使用计算机系统提供手段。操作系统管理的对象中,硬件资源有 CPU、内存、寄存器、堆栈、辅助存储器和输入输出设备等,而软件资源有系统软件、应用软件和数据等。总而言之,操作系统的功能包括如下5部分:处理器管理、存储器管理、设备管理、文件管理和用户接口。

1.2.1　处理机管理

处理机管理就是对 CPU 进行管理,即如何分配处理机?当系统中存在多个程序要运行时分配给谁?分配多长时间?何时收回?等等。众所周知,CPU 是计算机系统的核心,是最宝贵的硬件资源;如何调度程序以使 CPU 尽可能地忙起来,减少其空闲时间,提高其利用率,相对提高系统的处理能力,是操作系统所要重点解决的问题。例如,当某个用户程序 A 进行输入输出操作时,此时 CPU 处于空闲状态,是否可以将 CPU 暂时分配给用户程序 B 并运行;当用户程序 A 在输入输出操作完成时再中断用户程序 B 并返回到用户程序 A 的断点处继续执行,从而减少 CPU 的空转时间,提高 CPU 的利用率。

为了方便处理机的调度、分配和管理,引入了进程和线程的概念。进程是对处于运行状态下的程序的动态描述,而线程是进程内部的一个控制流;这些概念的具体含义在以后的相关章节中再详细解释。有了上述概念后,就可以引出多任务、多进程和多线程的概念,即在一定时间内(宏观上)计算机同时执行多个任务、多个进程和多个线程;对于单 CPU 系统,由于仅有一个 CPU,某一时刻(即微观上)不可能有多个任务或多个进程同时占有 CPU,所以只能采用某种调度策略将 CPU 轮流分配给各任务或进程,使它们在一段时间间隔内均能得到执行的机会,从而实现宏观上的同时运行(称之为并发执行)。

处理机的管理最终可以归结为对进程的管理,因为处理机的调度、分配均是以进程为基本单位的,这一部分还包括进程管理、同步、互斥、通信和死锁等内容。

1.2.2　存储管理

存储器管理的对象为主存储器,简称为内存或主存;因受成本等方面条件的限制,其容量有限。众所周知,程序若要运行需首先装入内存,然后待分配到 CPU 等必要的系统资源后便可以执行了;那么用户程序如何申请到内存?申请到内存后又怎样从外存(磁盘等)装入内存?具体装入到内存的哪个位置等?是否与内存中已有的程序发生冲突?如果这些问题均需用户来考虑,用户编程的难度简直不可想象;如果上述功能可以通过软件自动实现,则可以为用户减轻很大的负担;出于上述考虑,操作系统中设计了存储器管理模块,用来实现存储器的自动管理和高效使用。

存储器管理是对内存资源进行管理,功能之一是为位于外存中的程序分配一定的内存

空间，并将之装入内存，此时存在3种情况：

（1）若内存中有足够的空闲空间，程序便可以顺利装入内存。

（2）当要装入内存中的程序大于空闲内存空间的大小时，还需根据一定的算法将内存中的某些程序交换到外存，以空出内存空间。

（3）当大于内存总容量的某个大型程序需要运行或多个用户程序要同时并发执行时，此时内存全部处于空闲状态仍满足不了需要。在这种情况下就需要提供某种机制，仅将要运行的那部分程序装入内存，而其他部分暂时存在外存，在运行时再装入内存，从而实现小内存运行大程序的目的。这样不需扩大物理内存的容量，而是借助于软件等技术把内存和外存结合起来，从逻辑上扩充内存的相对容量，并通过存储器管理功能为用户提供透明服务（即用户无须知道如何分配内存以及程序具体放在内存的哪个位置等），使用户感觉到存在一个比实际内存空间大得多的存储空间供其使用，这就是虚拟存储技术。

由上述内容可以看出，存储器管理的主要任务是内存的分配和回收、内存扩充、存储共享与保护、地址变换等，并以提高内存的利用率、扩大内存的相对容量、为用户提供透明服务为目的。实现上述任务的具体内容如下：

（1）内存的分配和回收。即为要装入内存的程序分配内存空间，在程序运行结束后要释放所占用的内存资源，并由系统进行登记，以便分配给将要运行的程序。

（2）内存扩充。为了解决分配给作业或进程的内存空间不足的问题，达到小内存运行大程序的目的，同时也为了提高进程的并行性，使系统资源得以充分利用，均需要对内存的容量进行相对扩充。操作系统主要采用软件技术（如覆盖技术、交换技术和虚拟技术等）对存储系统进行管理，利用大容量的辅存来弥补内存空间的不足（详见第5章）。

（3）存储共享与保护。这里有两方面含义：一是指存储共享，即当多个程序包括一个公用程序段时，这个公用程序段在内存中仅存储一个副本，为一个共享程序段；存储管理要解决这一公用程序段的共享问题。二是指存储保护，即内存中存在多个运行程序时，存储管理要保证每个用户程序只能访问自己的存储空间，不能对驻留在内存中的操作系统等系统软件或其他应用程序造成破坏。存储保护通常需要硬件的支持，常用的方法有界限地址寄存器和存储保护键等。

（4）地址变换。当程序从外存装入内存时，需将程序中每条指令的地址（相对地址或逻辑地址）转换为内存中的地址（物理地址）。该过程可以在程序装入内存时完成（静态地址变换），也可以在程序执行时进行（动态地址变换），分别称为静态或动态地址重定位，通常需要硬件支持。

1.2.3 文件系统

计算机系统所处理的对象为程序和数据，这些程序和数据通常以文件的形式存放在磁盘、光盘、磁带等外部存储器上，需要时再装入内存。外存的存储容量相当大，存有大量的系统软件、应用程序和数据等，这些文件在外存中如何组织才能方便用户对文件的查找和透明存取均是文件管理模块所要解决的问题。实现对文件的“按名存取”是操作系统所要完成的另一项重要功能，即用户只要知道文件名就可以方便地存取文件，具体如何组织和查找文件均由文件管理模块来完成，用户无须关心。此外，文件管理还应提供文件的共享和保护、文件操作等功能。其中，文件的共享对于多道程序系统、多用户系统或多任务系统是相当重要

的,它使得被多个程序共享的对象在外存上仅保留一个备份,从而提高了外部存储器的利用率;文件保护功能用于对外部存储器中的文件进行保护,防止文件受到其他用户非法读取或破坏;文件操作为用户对文件进行操作提供手段,如文件的读、写、复制和删除等。此外,文件系统还应提供磁盘的容错功能,以保证所存储文件的可靠性。

1.2.4 设备管理

设备管理是指计算机中除了 CPU 和内存以外的输入输出设备的管理。输入输出设备也称为外部设备,其种类繁多,功能差异很大,使用方法各异。设备管理的主要任务就是通过驱动程序和控制程序自动实现对输入输出设备的管理和调度以及数据传输,以使用户不必了解输入输出设备的具体细节就可以完成输入输出操作,实现输入输出操作与设备的无关性。设备管理的另一任务就是通过中断技术、DMA(Direct Memory Access)技术、通道技术和缓冲技术等使 CPU 与外部设备能够并行、高效地工作,解决高速 CPU 与低速外设间速度不匹配的矛盾,从而提高系统资源的利用率。

由此可见,设备管理应包括如下功能:响应用户进程提出的 I/O 请求;为用户进程分配 I/O 设备;对系统的 I/O 设备进行管理;控制 CPU 与外设间的数据交换。

1.2.5 用户接口

操作系统作为计算机系统资源的直接管理者,应提供相应的手段供用户使用,以使用户可以方便地使用系统的各种资源,这种手段就是用户接口。用户接口通常有两种,一种供用户通过键盘和鼠标直接对计算机进行操作时使用,称为操作级接口;另一种供用户在编程过程中使用,称为编程级接口。这两种接口体现为 3 种形式:命令接口、图形接口和编程接口。

命令接口和图形接口为用户提供了直接控制计算机或使用系统资源的手段。命令接口为用户提供若干条命令,用户通过键盘和显示器完成想要的操作。用户所要完成的操作可以由若干条命令组成;这些命令可以通过键盘逐条输入系统,这种方式称为联机用户接口;这些命令也可以通过编辑程序录入到计算机内,形成一个文件,系统通过运行这个文件来完成指定的功能,这种方式称为脱机用户接口,如批处理等。图形接口(Graphics User Interface,GUI)为用户提供了功能图标或菜单并显示在显示器上,用户若想完成某项功能,就可以用鼠标点击相应的图标或者选择相应的菜单,如现在流行使用的视窗操作系统 Windows 系列、Linux 等均提供这样的方式。

编程接口是为用户程序在执行过程中访问系统资源而设置的,是用户程序访问系统资源的唯一手段。操作系统通过系统功能调用来实现它的编程接口。开发软件时,用户通过编辑程序将系统功能调用嵌入到程序中来访问系统的软硬件资源,从而对用户屏蔽了功能实现的具体细节。实际上命令控制界面也是通过系统功能调用来实现的。每个系统调用具有不同的功能号,由专门的指令嵌入到程序中以完成特定的功能。对于不同的操作系统有不同的系统调用,一般有几十至几百条。系统调用可分为如下几类:设备管理、文件管理、进程管理、存储管理和线程管理。

操作系统设计的目标之一就是为用户提供直观、简单、高效、易用、透明的用户接口。

随着操作系统的发展,还会出现新的用户接口形式,如语音式、触摸式等。

1.3 操作系统的发展

自20世纪50年代中期第一个批处理操作系统问世以来，操作系统已有50多年的历史，在此期间计算机硬件性能的逐渐增强以及用户需求的不断增加有力地促进了操作系统的发展；而操作系统的发展反过来不仅拓宽了计算机的应用领域，也促进了自身的不断完善。

1.3.1 手工阶段

从1946年问世的第一台计算机至20世纪50年代中期为第一代计算机时期，其主要组成元件为电子管，速度较慢；此时计算机没有操作系统软件，仅能满足单个用户独占使用，即计算机中仅有一道程序。运行程序时需用户通过输入设备(纸带输入机或读卡机)经手工操作将程序和数据输入进计算机并启动运行；运行结束后用户取走打印结果，然后下一个用户才能使用；这样程序运行的过程也就是用户不断干预的过程；此时由于计算机的运算速度比较慢，相对于程序运行时间来讲，用户干预的时间所占的比例并不显著。

1.3.2 早期批处理

20世纪50年代中期至60年代中期电子管逐步被晶体管所取代，计算机由此发展到了第二代。此时计算机CPU的运算速度显著提高，手工干预和外部设备的慢速性与计算机CPU运算速度的快速性之间的矛盾日益突出；传统手工干预的操作方式严重妨碍了计算机资源利用率的提高，特别是CPU的利用率。在这种情况下引入了批处理的概念，即各用户将其程序和数据交给操作员组成一批作业，并由操作员统一装到输入设备(纸带输入机或读卡机)上，然后由监督程序依次调入内存并启动处理，打印输出结果后监督程序再自动选择下一个作业并启动运行，直到处理完这批作业。显然在这批作业的处理过程中作业的装入、启动等均由监督程序自动完成，无须操作员或用户的干预，从而提高了系统资源的利用率。此时的监督程序即为操作系统的雏形，由于计算机内仅有一道程序在运行，故称之为单道批处理系统；并且因为作业的输入输出、运行等均在同一台计算机上进行，故也称之为联机批处理系统。

由上面的论述可见，监督程序的引入免去了人工干预，提高了系统资源的利用率；但此时CPU和输入输出设备在速度上的不匹配问题便暴露出来了，特别是在输入输出过程中CPU一直处于等待状态。为了解决上述矛盾提出了脱机批处理的概念，即输入输出工作在一台单独的计算机(称为卫星机或外围处理机)上进行，而程序运行在另一台主处理机上。此时操作员通过外围处理机将程序和数据存储在磁带等较快速的存储介质上，然后将该存储介质与主处理机连上，并由监督程序依次装入内存进行处理，计算结果也存在磁带等输出带上，待这批作业处理完后再取下连到外围处理机上进行输出；脱机批处理的过程如图1.2所示。这种脱机批处理仍属于单道批处理系统，但它很好地缓解了CPU的快速性与输入输出设备的慢速性之间的矛盾，在一定程度上提高了CPU等系统资源的利用率。

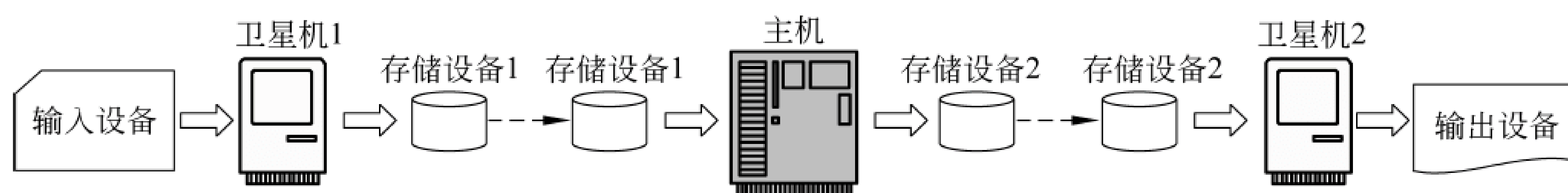


图 1.2 脱机输入输出流程图

1.3.3 多道程序系统

到了 20 世纪 60 年代初期,出现了对计算机技术发展具有重要影响的两种技术:通道技术和中断技术。通道为用于完成输入输出功能的专用处理单元,它通过某种方式和主处理机连接在一起;当主处理机处理的作业需要进行输入输出操作时,它就将必要的信息送给通道并启动它完成相应的输入输出操作,然后由监督程序暂停当前作业的运行并调度另一个作业投入运行;当通道完成输入输出操作后,它发中断给主处理机通知其所要的数据已准备好,此时主处理机再中断正在运行的作业并返回到刚被中断的作业继续执行。这样就可实现主处理机和外部输入输出设备间的并行工作,从而大幅度地提高了计算机系统资源的利用率。该系统的组成如图 1.3 所示。显然,此时主处理器内存中有多个作业,即存在多道程序,故称为多道批处理系统,相应的程序设计技术称为多道程序设计。在这种系统中,用户所提交的多个作业以队列形式在外存中排队,称为后备队列;在运行过程中由作业调度程序按照一定的算法依次将一个或多个作业调入内存,然后插入到就绪队列中,等待进程调度程序分配 CPU。这种系统通过作业类型的合理安排可以提高 CPU、内存和 I/O 设备等系统资源的利用率。如将计算型作业和 I/O 型作业均衡、交叉地安排在一批作业中进行处理,可以保证 CPU 和 I/O 设备同时处于运行状态,减少 CPU 和 I/O 设备的相互等待时间,从而提高这些系统资源的利用率。

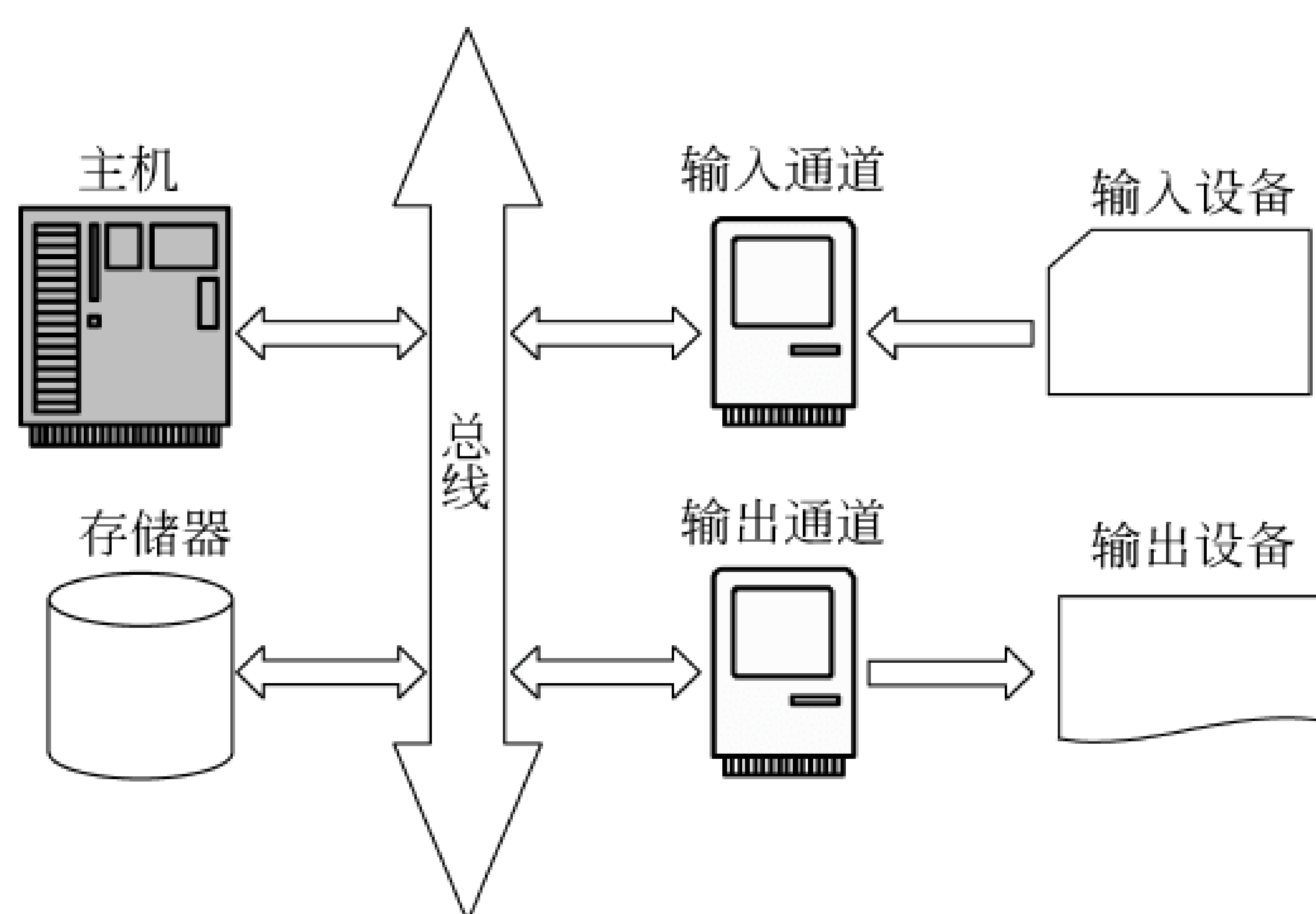


图 1.3 输入输出的通道方式

1.3.4 分时系统

随着多道程序设计技术的出现,此时的操作系统已逐步完善,且具有系统资源利用率高、作业吞吐量大等优点,并成为 3 种基本操作系统类型之一;但在程序运行过程中,用户难以直接干预,不能进行人机交互,不利于程序的联机调试;同时需等待一批作业执行完才能

得到执行结果，作业周转时间较长。为了改善批处理系统的性能，加上因为当时的计算机非常昂贵，希望一台计算机可以供多个用户同时使用，所以出现了分时操作系统。这种系统的计算机连接多个终端用户，各用户通过终端按一定的时间片轮流占用 CPU 等系统资源。相对于较慢的人工干预过程而言，对于较快速的计算机来讲，用户几乎感觉不到系统的响应延迟。最初问世的这种分时操作系统之一是由 MIT 开发、运行在 IBM 709 上的 CTSS (Compatible Time-Sharing System)，后来又移植到 IBM 7094 机上，最多可支持 32 个用户同时使用。

1.3.5 实时系统

所谓“实时”是指“及时”，实时系统是指系统能及时(或即时)响应外部事件的请求，在规定的时间内完成对该事件的处理，并控制所有实时任务协调一致地运行。

20 世纪 60 年代中期计算机进入第三代，计算机的性能和可靠性有了很大提高，造价也大幅度下降，使得计算机应用越来越广泛。在这一时期，由于计算机被用于工业过程控制、军事实时控制等，从而形成了各种实时系统。实时操作系统是以在允许时间范围之内做出响应为特征的。它要求计算机对外来信息能以足够快的速度进行处理，并在被控制的对象允许时间范围内做出快速响应。

1.3.6 通用操作系统

多道批处理系统和分时系统的不断改进以及实时系统的出现及其应用日益广泛，致使操作系统日益完善。在此基础上，出现了通用操作系统。它可以同时兼有多道批处理、分时、实时处理的功能，或其中两种以上的功能。例如，将实时处理和批处理相结合构成实时批处理系统。在这样的系统中，它首先保证优先处理任务，插空进行批作业处理。通常把实时任务称为前台作业，批作业称为后台作业。将批处理和分时处理相结合可构成分时批处理系统。

在保证分时用户的前提下，没有分时用户时可进行批量作业的处理。同样，分时用户和批处理作业可按前后台方式处理。

从 20 世纪 60 年代中期开始，国际上开始研制大型通用操作系统。这些系统试图达到功能齐全、可适应各种应用范围和操作方式变化多端的环境的目标。但是这些系统本身很庞大，不仅付出了巨大的代价，而且由于系统过于复杂和庞大，在解决其可靠性、可维护性、可理解性和开放性等方面都遇到很大的困难。但也有成功之例，UNIX 操作系统就是一个典型。

1.3.7 多种操作系统并存

20 世纪 80 年代以后，随着大规模集成电路制造技术的发展，迎来了个人计算机时代。1981 年 8 月微软公司公布了 DOS 1.0 版本的单用户操作系统，它运行在 8086 微处理器环境下。随后 CPU 由 8086、80286 发展到 80586 乃至奔腾系列，操作系统也随之迅速发展到 Window Vista、Window 7 和 Window 8。特别是近几年来，随着硬件技术的发展以及多媒体、Internet 与 Web 访问、集群计算等新的应用需求的不断提出，操作系统的性质发生了根本变化，这种变化不是简单地调整或增强现有操作系统的功能，而是采用微内核、多线程、对称多处理、分布式和面向对象设计等技术来组织、设计操作系统，进而出现了单用户多任务

操作系统、并行操作系统、分布式操作系统、网络操作系统和多媒体操作系统等,很好地适应了计算机硬件技术和体系结构的不断发展,满足了各个领域用户的需求。

1.4 操作系统的类型

在计算机技术发展的不同时期产生了不同种类的操作系统,以满足该时期特定的硬件条件下用户对计算机的使用需求。同时针对不同的应用场合,也开发了相应的操作系统以满足用户的特殊需求。

1.4.1 批处理操作系统

早期的计算机系统非常昂贵,为了能充分利用,应尽量保持计算机系统处于连续运行状态。此时用户不直接接触计算机,而是将由程序和数据组成的用户作业提交给系统操作员。当接收多个用户作业后,系统操作员将它们成批地装入计算机,然后由操作系统进行组织并按照一定的算法选择一道作业装入系统内运行;作业选择算法的优劣一般以作业平均周转时间等作为评判标准。在多道程序设计技术出现以后,批处理操作系统由单道批处理发展成多道批处理。

早期的批处理操作系统是单道的,一个作业单独装入系统并独占系统的所有资源,直到其运行结束后下一个作业才能装入系统。此时 CPU 等系统资源的利用率较低,特别是对于 I/O 操作较多的作业。为了提高 CPU 的利用率,引入了多道程序设计(multiprogramming)技术,即内存中同时装入多个作业;它们通过一定的调度算法轮流占有 CPU,这样可以使 CPU 尽量地处于运行状态,显著提高了 CPU 的利用率。

批处理操作系统一般运行在较大型的计算机系统上,它的特点是用户脱机使用计算机,多道程序同时运行、成批处理,目的是为了提高设备利用率和作业吞吐率。其缺点为作业周转时间长,无交互性,用户无法直接进行必要的干预。

1.4.2 分时操作系统

批处理操作系统环境下,用户以脱机方式工作,不直接和计算机接触,如果程序在运行过程中出现问题,用户不能直接干预,这样使用起来非常不方便,为此产生了分时操作系统。

分时操作系统将规定的 CPU 时间划分成若干个时间段,每个时间段称为时间片,然后按时间片将 CPU 轮流分配给系统内的每个程序。这样连接到计算机的每个用户或系统内的每个程序在系统运行期间均能得到服务。当该时间片选择合理或计算机的速度足够快时,各用户或程序并没有感觉到其他用户或程序的存在,好像它在独享计算机系统的所有资源,而实际上是各个用户或程序轮流地使用计算机。显然,此时一台计算机可以有多个终端用户,各用户在各自的时间片内能够和计算机进行交互作用,系统对用户的要求能够及时响应。

分时操作系统具有多路性、独立性、及时性和交互性等特征。多路性指宏观上多个用户同时使用计算机,而微观上是各用户轮流使用。独立性是指各用户均感觉是独占计算机的所有资源,而感觉不到其他用户的存在,各自完成各自的任务。及时性是指计算机对各用户的请求均能及时响应。交互性是指用户能够和计算机进行人机交互,干预计算机的运行。

由此可见,分时系统是一个联机(on-line)、多用户(multi-user)和交互性(interactive)的

操作系统。

目前常用的通用操作系统为分时操作系统和批处理操作系统的结合。对响应时间要求较高的用户一般工作在分时状态下，称为前台；而对响应时间要求不高的作业一般工作在批处理状态下，称为后台，如打印作业。

1.4.3 实时操作系统

实时操作系统是指计算机能够及时响应外部事件的请求，在规定的时间内完成对事件的处理，并有效地控制被控对象和实时任务协调地运行。实时操作系统一般是为了满足某一类实时系统的应用而设计的，如工业生产线的控制、武器控制系统等；在设计上首先要保证它的实时性和可靠性，其次才是系统效率。它常有两种类型：实时控制系统和实时信息处理系统，前者能够实时采集测量数据，并对数据进行实时加工、处理和输出，这种系统主要用于军事和工业生产过程的自动控制。后者能够对用户的请求及时做出响应，并能及时修改和处理系统中的数据，主要用于银行业务、订票系统的实时事务管理。

实时操作系统在嵌入式计算得到了越来越广泛的应用。特别是移动计算等非PC、PDA(个人数字助理)和手机等新设备的出现，更加强了这一趋势。

例如，随着移动通信进入3G时代，诺基亚等公司研制的Symbian手机操作系统、微软公司研制的WindowsMobile、近年崛起的操作系统新秀Linux等都已有了巨大的市场和用户群体。

1.4.4 通用操作系统

批处理操作系统、分时操作系统和实时操作系统是操作系统的3种基本类型，在此基础上发展了具有多种类型操作系统特征的通用操作系统，它可以兼有批处理、分时和实时操作系统的部分特征，具有一定的通用性，能够适用于较宽的应用领域。

1.4.5 个人计算机操作系统

个人计算机操作系统一般运行于个人计算机环境下，是一种单用户、单任务或单用户、多任务的操作系统。它提供联机交互功能，而且采用图形用户界面，操作直观、友好，易学易用。这类操作系统具有虚存、并发、多任务、联网等特征或功能。这类操作系统有DOS、Windows 3.x/95/98/2000/2003/Vista/7/8、OS/2和Linux等。

1.4.6 嵌入式操作系统

嵌入式操作系统是运行在嵌入式系统环境中，对整个嵌入式系统以及它所操作、控制的对象进行有效管理的系统软件。嵌入式系统是以应用为中心，软硬件可裁减，能够满足应用系统对功能、可靠性、成本、体积和功耗等综合性要求的专用计算机系统；它主要由嵌入式处理器、相关支撑硬件、嵌入式操作系统及应用软件系统等组成，它是集软硬件于一体的可独立工作的“器件”；其中嵌入式操作系统为嵌入式系统的灵魂，相当于常规操作系统的内核(实现操作系统功能中最基本的部分，也称微内核)，它负责嵌入式系统全部软、硬件资源的分配、调度和管理，控制并协调并发活动，具有小巧、实时性强、可装卸、代码可固化、弱交互性、高稳定性和接口规范等特点。这种操作系统按实时性能可以分成两类：一类是面向控

制、通信等领域的强实时性嵌入式操作系统,如 WindRiver 公司的 VxWorks、ISI 的 pSOS、ATI 的 Nucleus 等;另一类是面向普通电子产品的弱实时嵌入式操作系统,较著名的为 Windows CE、Palm OS 等,这类电子产品包括个人数字助理(Personal Digital Assistant, PDA)、移动电话和机顶盒等。

1.4.7 网络操作系统

网络操作系统运行于计算机网络环境下,它能够利用局域网低层提供的数据传输能力,为高层网络用户提供通信服务以及资源共享等功能,它是按照网络体系结构协议标准开发的操作系统。除了实现常规操作系统的功能外,它还包括如下功能:

(1) 协调用户。对系统资源进行合理分配和调度。

(2) 提供网络互连和通信服务功能,能够将异构环境下的计算机等设备连接起来进行相互通信,使得用户可以访问网络上的软硬件资源,实现资源共享。

(3) 用户访问控制。对用户进行访问权限的设置,保证系统的安全性,提供可靠的保密方式。

(4) 文件管理。对网络环境下的海量级文件进行快速、准确、有效、安全可靠的管理。

(5) 网络系统管理。对网络资源进行管理,监示网络活动,进行网络流量监测和计费、通信负载平衡以及安全监测等。

网络操作系统具有多用户、多任务功能,网络用户只有通过网络操作系统才能享受计算机网络提供的各种服务。目前流行的网络操作系统有 Windows XP/7/8、UNIX 和 Linux 等。

1.4.8 并行操作系统

并行计算机系统一般包括并行工作的多台处理机,这些处理机经总线或高速通信网络系统连接起来,机间通过共享内存、收发消息等方式进行通信,各台处理机上可以运行不同的进程或线程,真正实现多进程(线程)的并行执行。这种系统的组成硬件一般要求相同,并由统一的全局操作系统进行管理,故也称为紧耦合系统。在这种硬件环境下运行的操作系统就是并行操作系统,它的显著特征就是并行性,即将系统内运行的多个进程或线程分派到不同的处理机上同时运行。该操作系统还具有负载平衡功能,能够根据各台处理机的忙闲情况和运算能力合理地分配任务;此外,由于各处理机上均有自己的 Cache,各 Cache 信息的一致性问题也是并行操作系统所要考虑的一个重要问题。并行操作系统往往依赖于并行计算机系统的体系结构,目前较成熟的并行计算机系统有对称多处理机系统(Symmetrical Multi-Processors, SMP)和大规模并行处理系统 (Massively Parallel Processors, MPP),其中对称多处理器操作系统是较为成熟的一种并行操作系统。

1.4.9 分布式操作系统

运行在分布式系统环境下的操作系统称为分布式操作系统。分布式系统(distributed system)是一些独立计算机的集合,这些计算机通过通信网络等手段连接起来,并由分布式操作系统来统一协调管理,用户在本地计算机上就可以透明地使用分布式系统中的各种资源,获得比单机系统更强的处理能力。分布式系统是建立在计算机网络基础之上的松散耦

合系统，而计算机网络中的设备可以是异构的，所以分布式系统环境下的资源管理是相当复杂的。分布式系统的功能之一是能够同时处理多个任务（或一个任务的多个子任务），这些任务可以分配到多台计算机上完成，这就涉及任务分派，在任务分派过程中尽量能充分发挥每台计算机的效率，并使它们承担与其计算能力相称的运算量，此功能称为负载平衡。当运行某个进程的计算机出现了故障，或者其他计算机上的计算资源更适合该进程运行，这个进程就需要迁移到另外一台计算机上来运行，进而分布式操作系统要具有进程迁移功能。此外，分布式操作系统还要完成分布式进程的同步与控制、计算机间的进程通信等功能。集群系统（cluster system）是分布式系统的一个典型例子，它通过低档的计算机、网络互联设备和集群操作系统构造出性能相当于超级计算机的集群系统。

由上面的分析可见，分布式操作系统通过通信网络将物理上分散的具有自治功能的计算机系统连接起来，实现全系统的资源分配、任务划分和调度、信息传递和通信以及资源共享等功能，使系统内的多台计算机用分工协作的方法高效地完成各种不同的任务。该类操作系统侧重于任务的分布性和各节点间的协作性，即将一个大任务按照一定的算法划分为若干个子任务，并将其分派到不同的处理节点上进行处理。它具有强健的任务分解分派算法和节点间动态负载的平衡能力。它具有处理能力强、可靠性高等特点。目前，分布式操作系统尚处于研究实验阶段，还没有真正成熟的实用系统。

1.4.10 多媒体操作系统

传统的操作系统管理的对象是普通的数字和字符等，而多媒体操作系统所管理的对象除包括这些具有明显的结构化特征的数据信息外，还要能够处理声、图、文字等多媒体数据以及其他一些半结构化或非结构化信息。这些多媒体数据信息量大，属于海量储存；而且多种媒体数据间往往存在一定的联系，处理多个数据流的多个进程常常需要同步，并要求在限定的时间内完成，如视频点播（Video On Demand，VOD）和视频会议等。处理对象的特殊性要求多媒体操作系统在进程调度、数据的组织和存取、磁盘管理和调度等方面不同于传统的操作系统。通常，多媒体操作系统在传统的操作系统基础上增加了 Audio、Video 和 CD-ROM 等驱动程序，并增强了系统的实时性以适应多媒体数据读取和传输的要求。

1.5 操作系统的基本特征

操作系统的种类较多，每种操作系统各有其特征，但它们均具有并发、共享、虚拟和异步4个基本特征，其中并发和共享是操作系统最基本的特征，其他特征都是以它们为前提的。

1.5.1 并发性

并发（concurrence）是指两个或多个事件在同一时间间隔内发生。并行是与其相似的另一个概念，它是指两个或多个事件在同一时刻同时发生。在多道程序环境下，并发性是指在一段时间内，宏观上有多个程序在同时运行。此时对于单处理器系统而言，每一时刻只能有一道程序在运行，微观上多道程序只能分时地交替执行。而对于多处理器系统，可并发执行的程序可以被分配到多个处理机上，实现它们的并行处理。并发机制的引入有效改善了系统资源的利用率，提高了系统的处理能力，但使系统的管理变得更为复杂，如位于系统内

的多个程序如何分配资源,如何切换运行状态,在其运行过程中如何不对其他程序的运行环境造成影响等。

1.5.2 共享性

共享(sharing)是指系统资源可供内存中多个并发执行的程序同时使用,目的是提高系统资源的利用率。系统资源指CPU、内存、数据和各种外设等。资源共享有两种方式:互斥共享和同时共享。

互斥共享是指系统的某些资源,虽然它们可以被多个程序共同使用,但在某一段时间内它们仅允许一个程序使用,只有等到该程序使用完后其他程序才能使用,这种资源称为临界资源或独占资源。计算机系统的大多数物理设备以及某些软件中所用的栈、变量、表等均可能为临界资源,它们要求互斥使用。打印机是互斥类共享资源的一个典型例子。

同时共享是指另一类共享资源,这些资源允许在一段时间内被多个程序“同时”使用。这里的同时仍是一个宏观概念,而实际上是各程序交替地使用这些资源。如对磁盘的访问(读/写)即同时共享操作。

1.5.3 虚拟性

虚拟性(virtual)是将一台物理设备映射成多个逻辑设备,并分配给每个用户,使得各用户认为他独自占有一台实际的物理设备,如分时操作系统环境下的CPU和处于共享状态的打印机。虚拟性表现的另一方面是操作系统中普遍采用的虚拟存储技术,该技术将主存储器和辅助存储器有机地结合起来,通过软件技术为用户提供了无限大的虚拟存储空间,可以运行比内存空间大得多的程序,但程序运行的实际存储空间仅为有限的内存空间,而用户感觉不到这种有限内存空间的局限性。

1.5.4 不确定性

在多道程序环境下,允许多个进程并发执行,但只有进程获得了所需的资源后才能执行。由于系统资源的共享性和有限性,当某一进程在运行过程中因申请的资源正在被其他进程使用,则该进程只能暂停执行,直到该资源被释放才能继续运行。如某个进程运行过程中要使用打印机,而是此时打印机正在被其他进程使用,则该进程只能暂停,进入等待状态。这样,进程的运行在系统中呈现为运行、暂停、运行、暂停、…,而且进程状态的转变受系统资源使用情况的制约,具有一定的不确定性(uncertainty)。由此可见,进程的执行是以不可预见的速度向前推进,是异步(asynchronism)的。

操作系统的不确定性是指程序执行过程中的不可预测性,并不是指程序结果的不确定性。程序执行结果的不确定性的原因往往是指由于程序设计本身的错误或由于用户各程序之间的配合(如同步、互斥)不当引起的。

1.6 操作系统的组成结构

操作系统是一个十分复杂、庞大的系统软件,是完成操作系统功能的各种系统程序的集成。那么这些系统程序是按什么样的结构组织在一起的呢?这就是操作系统的组成结构。

在操作系统的发展历程中,对其结构的研究一直是计算机领域所关注的一个重点。而程序设计技术以及计算机体系结构的不断发展促使了操作系统结构的不断演变。目前,具有传统结构的操作系统已发展成为具有微内核结构的现代操作系统。

1.6.1　无结构的操作系统

在操作系统发展的初期,计算机硬件资源有限,配置较低,特别是内存容量很小,通常只有几万字节,CPU 的速度也很慢;此时,操作系统的开发只注重功能如何实现和如何提高系统的效率,而没有考虑采用统一的标准和结构来设计、开发操作系统。此时程序设计的技巧仅限于如何编制紧凑的程序,以便有效地利用有限的内存空间,而对 GOTO 语句的使用不加任何限制,所设计的操作系统为若干个功能程序的简单聚集,缺乏清晰的结构,所以该阶段的操作系统是无结构的。

1.6.2　模块化结构的操作系统

20 世纪 60 年代出现了模块化程序设计技术,它通过任务分解和模块化来降低开发大型软件的复杂程度。如前所述,操作系统是一个非常复杂的大型软件,引入模块化程序设计技术后,操作系统软件按功能划分为若干个相对独立的程序模块,每个模块用于实现某种资源的管理和调度,如进程管理模块、存储器管理模块和设备管理模块等;各模块又可以划分为若干个子模块,如进程管理模块又划分为进程调度、进程同步和进程通信等子模块,每个子模块完成更为具体的功能;当某个子模块较大时,还可以进一步细分。模块化操作系统的组成如图 1.4 所示。该系统的优点是系统的结构紧凑、接口简单、效率高,但也存在模块间数据联系过多、独立性差等缺点。

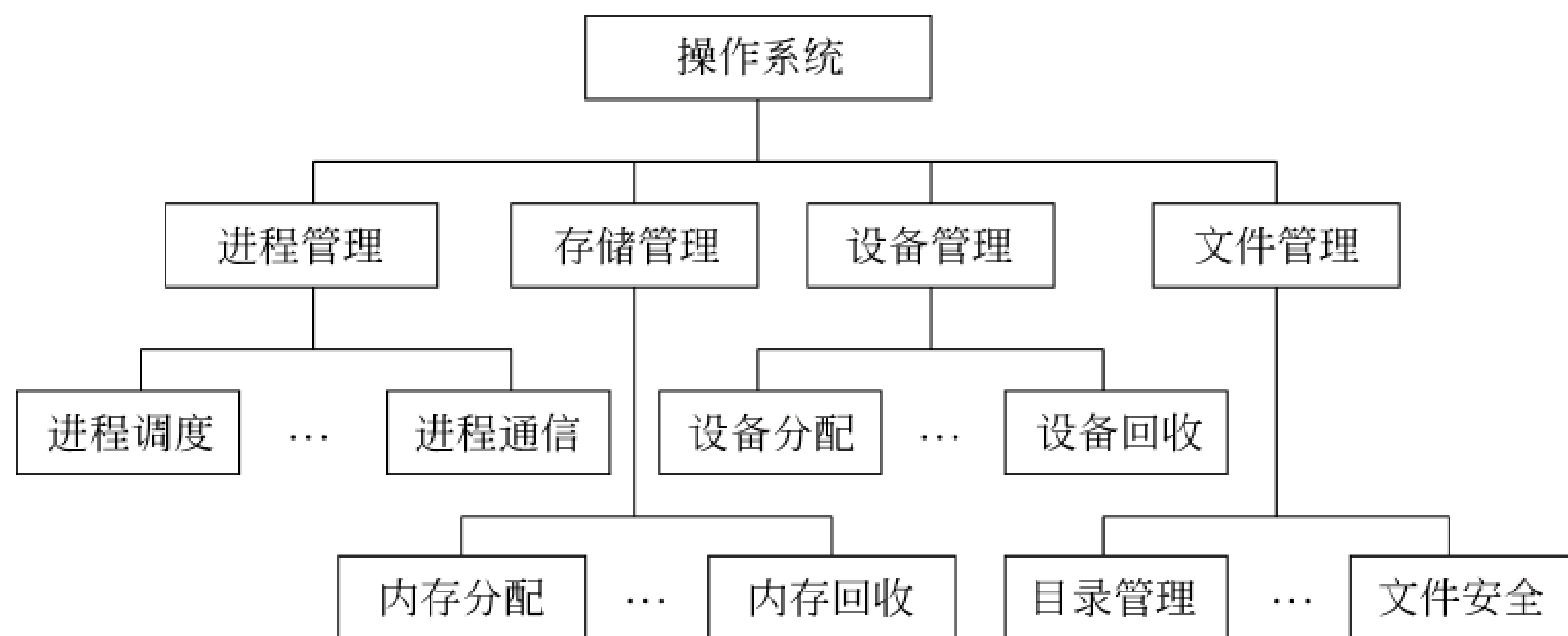


图 1.4　操作系统的模块化结构

1.6.3　分层结构的操作系统

随着计算机硬件性能指标的快速增长,操作系统的功能也越来越强,不但操作系统的规模越来越大,其复杂性也随之增加。对此操作系统的设计不仅采用了模块化技术而且也引入了层次式设计方法,即在硬件的基础上一层一层地向外扩充软件,其中低层软件为高层软件提供服务,高层软件通过调用低层软件来实现其功能;这样由于每一层仅使用其低层软件所提供的功能和服务,故减少了模块间的联系,增强了各模块的独立性,同时各模块采用统一的接口供上层调用,可以使系统的结构更加清晰和规范,系统的调试和验证也变得更为容

易。采用层次化设计方法实现的操作系统的典型结构如图1.5所示。

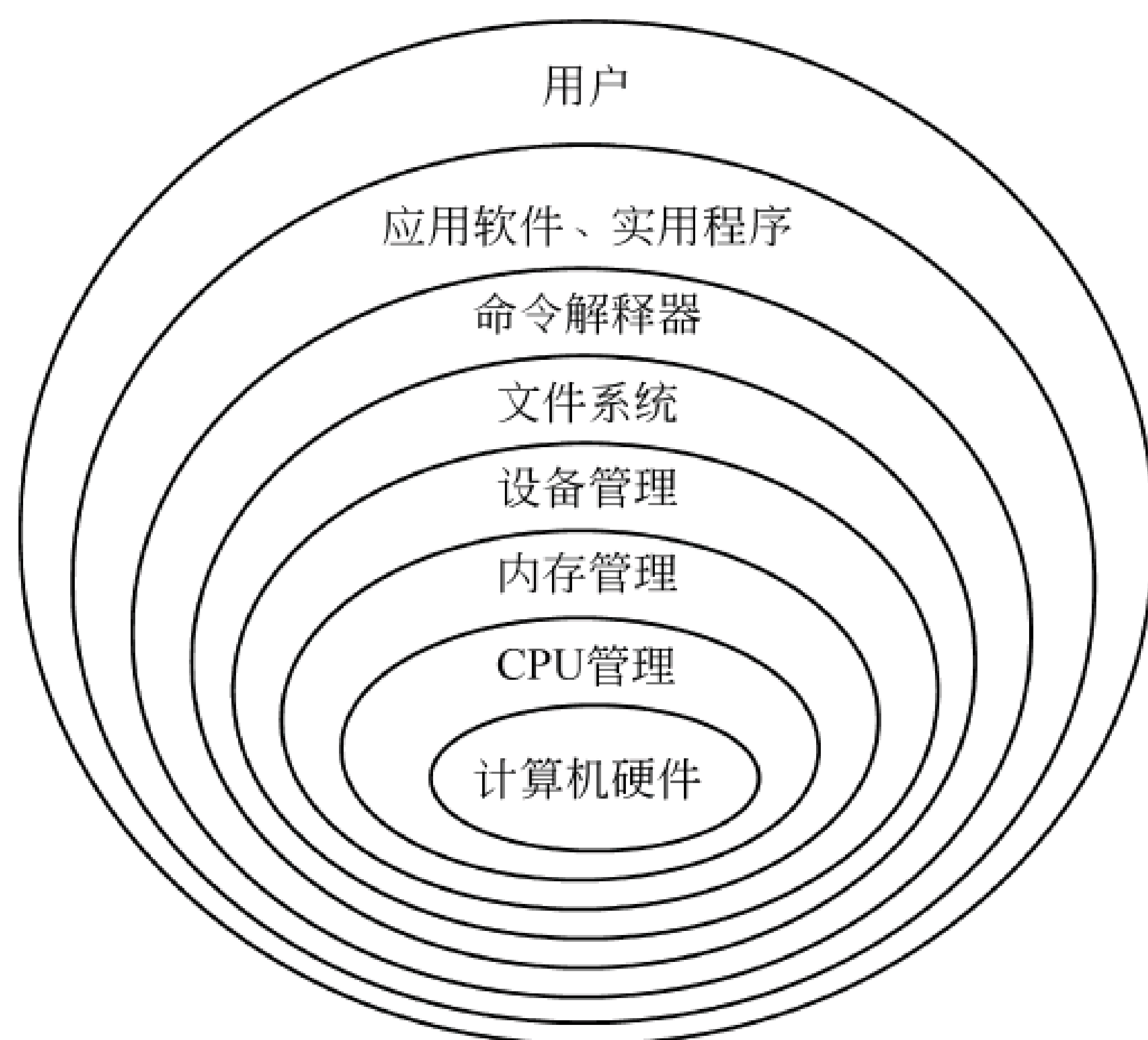


图1.5 操作系统的典型层次结构

1.6.4 微内核结构的操作系统

微内核(microkernel)是20世纪90年代发展起来的操作系统开发新技术。采用该技术开发的操作系统具有微内核结构,它能有效地支持多处理机运行,故非常适合于并行操作系统和分布式操作系统的开发,最典型的是客户/服务器系统(client/server system)。目前比较流行的、能支持多处理机运行的操作系统大都采用微内核结构,如当前广泛使用的Windows NT操作系统。

微内核的引入源于处理器性能的提高以及多处理器技术的发展。针对硬件性能的不断增强,人们对操作系统的要求越来越高,必须不断地在操作系统中增添新的功能,从而导致操作系统规模的急剧膨胀,从早期的几万字节到目前的几亿字节,使得操作系统的开发、维护和移植变得相当困难。为了减少操作系统的复杂性、增强其可扩展性和可维护性而产生了微内核技术。

所谓微内核技术是指精心设计的、能实现现代操作系统核心功能的小型内核,它与一般的操作系统不同,它短小精炼,不仅运行在核心态,而且开机后常驻内存。微内核中仅包括操作系统中最主要、最基本、最底层的功能,如进程管理、存储器管理、进程通信、低级I/O功能等,因而它不是一个完整的操作系统,它为通用操作系统的开发提供底层支持,在此内核的基础上和模块化、层次化设计方法以及面向对象技术相结合可以非常方便地开发出具有各种特点的操作系统。

1.7 研究操作系统的几种观点

操作系统是复杂的大型系统软件,它不但为应用人员提供使用计算机系统软、硬件资源的手段,同时它也是广大计算机科学与技术工作者研究、开发的对象。操作系统开发人员希望其所开发的系统功能强大、占用系统资源少、代码精练;而使用人员则希望操作系统界面

友好、使用方便、可移植性强。所以观察的角度不同，对操作系统的要求是不一样的。这样在研究操作系统的过程中，所站的角度不同，所持的观点也不尽相同。一般研究操作系统有如下几种观点：资源管理的观点、用户界面的观点和进程管理的观点。

1.7.1 资源管理的观点

现代计算机系统由CPU、存储器、输入输出设备以及程序和数据所组成。这些硬件和软件为系统软件和用户程序运行提供所需的各种资源和工作环境。操作系统的目的就是对这些资源进行有效的管理，协调这些资源的分配和使用，避免因有限资源竞争而导致系统不能正常运行的现象发生，最终达到有效、安全地分配资源、协调任务运行、充分发挥软硬件资源效率的目的。

从资源管理的观点研究、分析和设计操作系统包括以下内容。

（1）记录资源状态。利用各种数据结构记录各种资源的当前使用情况，包括正在使用的资源以及处于空闲状态的资源的详细信息。

（2）资源的分配和调度。根据系统的设计目标确定一组分配原则和调度算法，以决定将存储器、处理器等资源分配给哪一个程序，何时分配，分配多长时间，等等，并在资源分配表上进行登记，记录资源的使用情况。

（3）资源回收。当资源使用完后，要对资源进行回收，并在相应的资源表上进行登记，以便以后进行分配使用。

1.7.2 用户界面的观点

用户是操作系统的最终使用者，所以操作系统产品最终能否被广泛接受是由用户来决定的。从用户角度来看，操作系统的功能越强、界面越友好、使用越方便，肯定越受欢迎。作为操作系统的用户并不关心操作系统采用什么样的结构、采用什么样的先进技术以及其具体细节是如何实现的，用户也不关心操作系统的开发过程，而只注重开发结果，即最终呈现在用户面前的操作系统是个什么样子，它为用户提供什么样的人机接口或手段以达到简单、方便、高效、安全地使用各种系统软硬件资源的目的。

1.7.3 进程管理的观点

进程是操作系统中的一个重要概念，它是为了描述系统中程序的并发执行过程而引入的，它是程序执行过程的动态描述，为资源分配的基本单位。并发作为现代操作系统的一个重要特性，无论是从资源管理的观点，还是从用户界面的观点，都很难对程序的这种动态特性加以描述。并发性是指操作系统控制许多能并发执行的程序段，在多处理机环境下这些程序段可以真正地并行执行，而在单处理机环境下它们则是宏观并行而微观顺序执行；它们可以是毫不相关的程序段，完全独立地执行；也可能是存在一定的联系，相互之间因共享资源或互为因果关系而相互制约。程序的执行可能很顺利，也可能因等待某种资源而暂停，呈现出执行、暂停、执行……即程序的执行体现为一个非常复杂的动态过程和异步特性。进程作为独立运行的实体和资源分配的基本单位，其管理贯穿于操作系统的始终。

1.8 典型操作系统简介

目前最常用的操作系统有 Windows、UNIX 和 Linux，而 MS-DOS、Mac OS、OS/400 和 OS/2 等操作系统虽然在计算机技术的普及应用过程中起到了非常重要的作用，但均已逐渐被界面友好、功能强大的 Windows、UNIX 和 Linux 等现代操作系统所取代。下面简要介绍目前流行的几种典型的操作系统。

1.8.1 Windows 系列操作系统

1983 年美国微软公司开始研制 Windows 操作系统，至 20 世纪 90 年代，在个人计算机操作系统领域，微软公司的 Windows 系列操作系统已占有绝对的垄断地位。

1985 年微软公司陆续推出了 Windows 操作系统 1.0 和 2.0 版本，1990 年又推出了 Windows 3.x 系列，它们均为 16 位操作系统，但需运行在微软早期推出的磁盘操作系统 DOS(Disk Operating System)之上。1993 年和 1995 年微软公司分别推出了 32 位的 Windows NT 和 Windows 95 操作系统，并彻底摆脱了 DOS 的限制，提供了多任务和多线程机制，支持网络、多媒体和移动计算等功能，提供了桌面视窗窗口，操作直观、简单。其中 Windows NT 充分利用了 32 位微处理器等硬件的新特性，采用和实现了大量新技术，支持对称多处理、多线程、多个可装卸文件系统(MS-DOS FAT、OS\2 HPFS、CD-ROM CDFS 等)，内置网络和分布式计算、互操作性等。该操作系统具有良好的可扩充性、可移植性和兼容性。

1998 年 6 月微软公司推出了 Windows 98 操作系统；它是一个 32 位的操作系统，但仍支持 16 位程序的运行，而后的版本便不具有这种特性。Windows 98 较早期的 Windows 95 增加了许多新特点，如新颖的 Internet 功能、FAT32 文件系统、支持 DCOM 等。而后虽然又推出了 Windows Me，但其性能较 Windows 98 并没有显著改善。

2000 年微软公司推出了 Windows 2000 系列，有 Professional 和 Server 等多个版本。它构筑在 NT 技术之上，是一个先进的、基于 PC 的客户/服务器平台，为下一代 PC 的商务操作系统。它支持高达 64GB 的内存和多种网络传输协议，具有更为友好的图形用户界面，而且安装、配置系统以及浏览 Internet 更为方便。Windows XP 为 Windows 2000 的下一个版本，它将个人操作系统和商用操作系统合二为一，增强了操作系统的适应性和通用性，是目前颇受欢迎的操作系统之一。目前微软公司已对 Windows XP 的重大安全漏洞进行了弥补，并进一步升级，增强了许多新的功能，推出了 Windows XP SE 版本。最近推出的为 Windows 7/8 版本。Windows 系列操作系统的发展历史如图 1.6 所示。

1.8.2 UNIX 操作系统

UNIX 操作系统于 1969 年诞生于贝尔实验室，它是唯一一个应用范围从微机到巨型机的多用户操作系统。它存在大量的变种，如 XENIX、SUN Solaris、IBM AIX 和 HP UX 等，以及克隆产品，如 Mach 和 Linux。

UNIX 操作系统的普及和 C 语言的开发以及初期 UNIX 的免费使用是分不开的。1973 年，Ritchie 用他开发的 C 语言重写了 UNIX，使其可移植性大大增强，走出了 UNIX

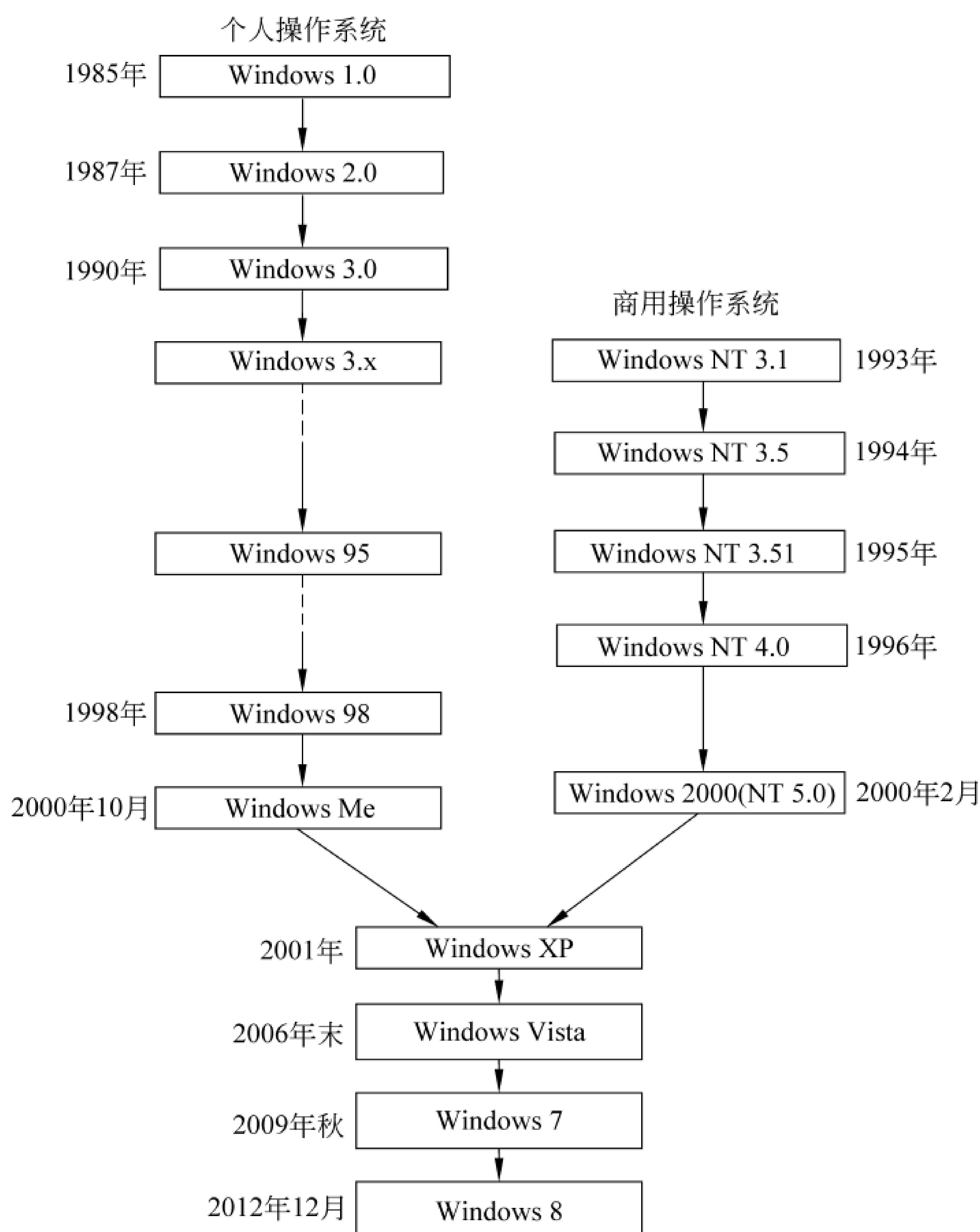


图 1.6 Windows 系列操作系统的发展历史

迈向成功的关键一步。1973 年至 20 世纪 70 年代末，UNIX 软件和源码免费公开使用，方便了各大学和研究机构对 UNIX 进行深入的研究、改进和移植。1978 年微软公司和 SCO 公司将 UNIX 移植到 Intel 8086，推出了 XENIX；同年 DEC 公司将 UNIX 委托移植到 VAX 机上，推出了 UNIX/32V。UNIX 系统由此逐渐形成了 3 条发展主线：由贝尔实验室发布的 UNIX 研究版、由加利福尼亚州大学伯克利分校发布的 BSD（Berkeley Software Distribution）和由贝尔实验室发布的 UNIX System Ⅲ和 System Ⅴ。到 20 世纪 80 年代，UNIX 已发展成为一种非常受欢迎的通用操作系统。

早期的 UNIX 操作系统具有内核结构简洁精练、接口简单规范、功能实用丰富、可移植性好、源代码免费开放等优点，但随着 UNIX 变种的增多，这些优点并没有被继承和发扬光大，反而朝其相反方面发展；加上后来微软 Windows 和 Linux 操作系统的出现，使得 UNIX 的应用受到了限制。但今天 UNIX 系统仍然是工作站、小型机和中型机以上各种类型计算机系统上的主流操作系统，其完美的技术内涵、系统的严谨性和安全可靠性等对现代操作系统的开发仍具有一定的指导意义。

UNIX 系统内部采用多用户、分时、多任务调度管理策略，文件系统可随意装卸，具有良

好的开放性和可移植性、强大的命令功能、完善的安全机制和网络特性。其问题简化和模块化设计技术已成为软件规范化设计和软件可重用理论的重要思想。

UNIX 操作系统的发展历程如图 1.7 所示。

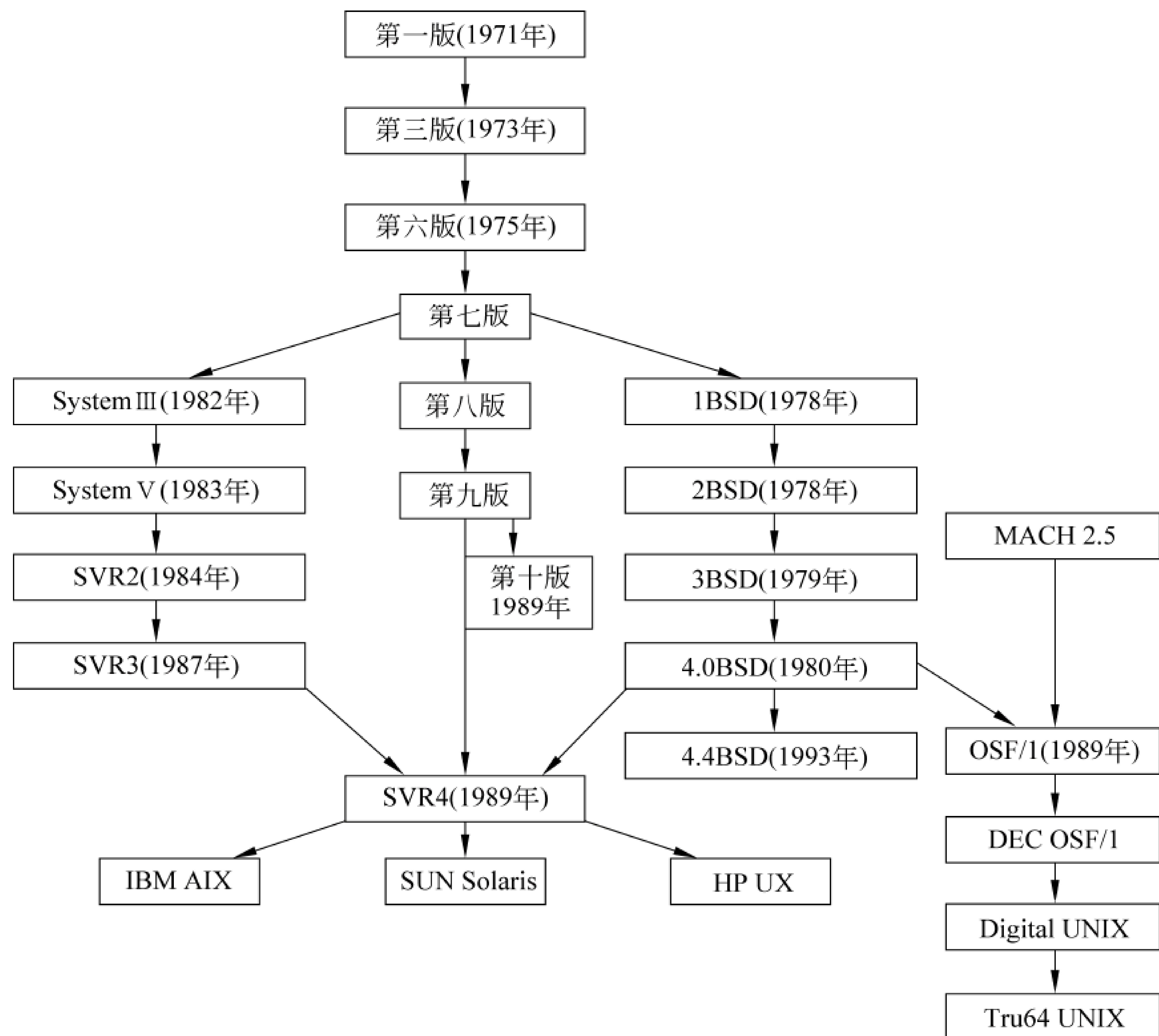


图 1.7　UNIX 操作系统的发展历程

1.8.3　Linux 操作系统

Linux 是运行于多种硬件平台、支持多种系统软件和应用软件、与 UNIX 兼容、符合可移植操作系统接口(Portable Operation System Interface，POSIX)标准、功能强大的操作系统。Linux 具有多用户、多任务、虚拟存储器和虚拟文件系统等特点，它是一种源代码开放、可免费使用的自由软件。Linux 网络功能相当强，目前大都运行在各种网络服务器上。

Linux 最初是由芬兰赫尔辛基大学计算机系学生 Linus Torvalds 在 1990 年出于个人兴趣开发的用于 Intel 386 个人计算机上的 UNIX 类操作系统。1993 年 Linus Torvalds 正式发布了 Linux 内核 V1.0。Linux 是在一般公共许可证(General Public License，GPL)保护下开发的自由软件。GPL 保证任何人有取得、修改、重新发布和免费使用自由软件的权利。Linux 虽然是 UNIX 类操作系统，但它没有使用一行 UNIX 的源代码。它继承了 UNIX 的许多优点，且在许多方面进行了改进。Linux 具有如下特征：

(1) 是真正的多用户、多任务操作系统，多个任务或多个用户可同时运行。

(2) 具有多线程功能，Linux 内核支持单个进程内存空间的多个独立线程控制。

(3) 在进程间采取内存保护,单个程序不可能击垮整个系统。

(4) 具有内核的编程接口,符合 POSIX 标准,可移植能力强,在源代码级上与 POSIX、System Ⅴ和 BSD 完全兼容。

(5) 提供具有内置安全措施的分层的文件系统,包括登录、口令、目录和文件权限。

(6) 提供 Shell 命令解释程序、Shell 编程语言以及其他多种高级编程语言。

(7) 具有图形用户界面,如 X-Window、Motif 等。

(8) 具有大量有用的实用程序和通信、联网工具,内置了许多网络服务功能。

(9) 运行于多种硬件平台,包括 Intel 系列微机,Power PC 系列、MIPS 系列以及 SPARC 系列工作站。

(10) 2.0 以后版本的 Linux 开始支持对称多处理器(Symmetric MultiProcessor,SMP)系统。

(11) 支持的文件系统多达 32 种。

(12) 通过各种仿真软件,Linux 系统可以运行 DOS、Windows 95/98 和 Windows NT 等操作系统的应用软件。

由于 Linux 是一个自由软件,它的版本比较复杂,其版本发展历程简单概括如下:

Linux 内核始于 1991 年由 Linus Torvalds 为他的 386 开发的一个类 Minix 的操作系统。Linus 初始曾想命名这个系统为 Freax,但很幸运的是最后他最后没有那样做。

Linux 1.0 的官方版发行于 1994 年 3 月,包含了 386 的官方支持,仅支持单 CPU 系统。

Linux 1.2 发行于 1995 年 3 月,它是第一个包含多平台(Alpha、Sparc 和 MIPS 等)支持的官方版本。

Linux 2.0 发行于 1996 年 6 月,包含很多新的平台支持,但是最重要的是,它是第一个支持 SMP(对称多处理器)体系的内核版本。

Linux 2.2 在 1999 年 1 月到来,它带来了 SMP 系统性能的极大提升,同时支持更多的硬件。

Linux 2.4 于 2001 年 1 月发布,它进一步提升了 SMP 系统的扩展性,同时它也集成了很多用于支持桌面系统的特性,如对 USB 和 PC 卡(PCMCIA)的支持以及内置的即插即用等。

Linux 2.6 于 2003 年 12 月发布,不仅包含了上述特性,同时也是一个无论对相当大的系统还是相当小的系统(如 PDA 等)的支持都有很大提升的“大跨越”。

Linux 发展的重要里程碑如下:

1990 年,Linus Torvalds 首次接触 MINIX。

1991 年,Linus Torvalds 开始在 MINIX 上编写各种驱动程序等操作系统内核组件。

1991 年底,Linus Torvalds 公开了 Linux 内核。

1993 年,Linux 1.0 版发行,Linux 转向 GPL 版权协议。

1994 年,Linux 的第一个商业发行版 Slackware 问世。

1996 年,美国国家标准技术局的计算机系统实验室确认 Linux 版本 1.2.13(由 Open Linux 公司打包)符合 POSIX 标准。

1999 年,Linux 的简体中文发行版问世。

2000 年,LinuxWorld China 2000 展览会召开,涌现大量基于 Linux 的嵌入式系统。

1991年10月5日,Linus Torvalds在新闻组comp.os.minix发布了大约有一万行代码的Linux v0.01版本。

到了1992年,大约有1000人在使用Linux,值得一提的是,他们基本上都属于真正意义上的黑客。

1993年,有100余名程序员参与了Linux内核代码编写/修改工作,其中核心组由5人组成,此时Linux 0.99的代码大约有十万行,用户大约有10万左右。

1994年3月,Linux 1.0发布,代码量为17万行,当时是按照完全自由免费的协议发布,随后正式采用GPL协议。至此,Linux的代码开发进入良性循环。很多系统管理员开始在自己的操作系统环境中尝试Linux,并将修改的代码提交给核心小组。由于拥有了丰富的操作系统平台,因而Linux的代码中也充实了对不同硬件系统的支持,极大地提高了跨平台移植性。

1995年,此时的Linux可在Intel、Digital以及Sun SPARC处理器上运行了,用户量也超过了50万,介绍Linux的Linux Journal杂志也发行了超过10万册之多。

1996年6月,Linux 2.0内核发布,此内核有大约40万行代码,并可以支持多个处理器。此时的Linux已经进入了实用阶段,全球大约有350万人使用。

1997年夏,大片《泰坦尼克号》在制作特效中使用的160台Alpha图形工作站中,有105台采用了Linux操作系统。

1998年是Linux迅猛发展的一年。1月,小红帽高级研发实验室成立,同年RedHat 5.0获得了InfoWorld的操作系统奖项。4月Mozilla代码发布,成为Linux图形界面上的王牌浏览器。RedHat宣布商业支持计划,网罗了多名优秀技术人员开始商业运作。王牌搜索引擎Google现身,采用的也是Linux服务器。值得一提的是,Oracle和Informix两家数据库厂商明确表示不支持Linux,这个决定给予了MySQL数据库充分的发展机会。同年10月,Intel和Netscape宣布小额投资红帽软件,这被业界视作Linux获得商业认同的信号。同月,微软公司在法国发布了反Linux公开信,这表明微软公司开始将Linux视为对手。同年12月,IBM公司发布了适用于Linux的文件系统AFS 3.5以及Jikes Java编辑器和Secure Mailer及DB2测试版,IBM公司的此番行为,可以看作是与Linux羞答答地第一次亲密接触。迫于Windows和Linux的压力,Sun公司逐渐开放了Java协议,并且在UltraSparc上支持Linux操作系统。1998年可以说是Linux与商业接触的一年。

Linux是开放的自由软件,其研究者遍布世界各地,他们之间通过计算机网络进行相互交流,所以Linux支持所有标准Internet协议(事实上Linux是第一个支持IPv6的操作系统)。由于Linux具有成本低、高可靠性、Internet应用软件丰富等特点,任何Linux发行版本都提供了电子邮件、文件传输和网络新闻等服务软件,所以它是Internet服务提供商(Internet Service Provider,ISP)所推荐的最流行的网络操作系统。而且,Linux是一个性能优异、标准的Web应用平台,利用它作为路由器、防火墙、Web服务器、电子邮件服务器、数据库服务器和目录服务器可以建立一个完善的、安全的Internet站点。

目前Linux系统有内核(Kernel)和发行套件两种版本。内核版本指的是在Linus领导下的开发小组开发出的系统内核的版本号,最近版本号为Linux-2.2.13、Linux-2.3.28和Linux-2.4。而一些组织机构或系统软件开发公司将Linux内核同应用软件和相应文档包装起来,并提供一些安装界面和系统设置管理工具,从而构成了一个发行版本,故Linux发

行套件也可以理解为以 Linux 为核心的操作系统软件包。常见的 Linux 发行套件有 RedHat、Debian、Slackware 和红旗 Linux 等。其中 RedHat 是 Linux 的一个非常重要的发行版本，由于支持多种硬件平台（包括 Intel、Alpha 和 Sparc）、集成的软件丰富以及友好的用户界面而颇受欢迎；RedHat 集成的软件非常完整，包括大量的 GNU 和自由软件；而且 RedHat 的系统安全性非常好，并提供快速的系统安全补丁。Debian 是由 GNU 发行的 Linux 版本，完全由网络上的 Linux 爱好者负责维护，被认为是最正宗的 Linux 版本，它的全部组件均为自由软件。红旗 Linux 是由中国科学院软件所等单位联合研制推出、具有自主知识产权的中文操作系统，与其他中文操作系统相比，它是世界上唯一一套在 Linux 上支持大字库（GBK）的操作系统。

Linux 的出现对 UNIX 在中低档服务器领域中的应用提供了有力支持，其优良的性能价格比有效抑制了 Windows NT 对 UNIX 服务器市场的冲击。据专家预测，将来的格局是传统的 UNIX 将主要占据高端服务器市场，而 Linux 与 NT 将分享中低档服务器市场，与此同时 Linux 将逐渐侵蚀 Windows 在客户端机市场上所占有的份额。

1.9 本章小结

操作系统是计算机系统中最重要的系统软件，它是最接近硬件的一层软件，任何其他软件均要运行在操作系统所构筑的软件平台之上，它显著地改善了计算机系统的易用性和使用效率。操作系统用于实现计算机系统软、硬件资源的管理，具体包括处理机管理、存储管理、设备管理、文件管理和用户接口等部分，其目的是为了提高计算机系统资源的利用率，并为用户提供直观、友好的使用接口，改善计算机系统的易用性。随着大规模集成电路制造工艺的快速发展、计算机体系结构的变革以及用户需求的不断增加，为操作系统的研究、设计和实现提出了许多新的课题，有力地刺激和加速了操作系统自身的不断完善和发展。目前，并行操作系统、分布式操作系统、网络操作系统和操作系统的安全性等已成为计算机科学与技术领域的重要研究课题，采用面向对象技术、具有微内核结构、支持多线程和对称多处理结构、具有开放性和分布式特点等已成为现代操作系统的显著特征。

本章最后简要介绍了 Windows、UNIX 和 Linux 三种有影响的操作系统。

习　题

1. 解释软件的种类及各种软件的功能。
2. 什么是操作系统？它的基本特征是什么？
3. 操作系统管理的对象是什么？设计操作系统的根本目的是什么？
4. 解释虚拟机的具体含义。
5. 操作系统的 5 种功能是什么？
6. 操作系统为用户提供了几种类型的接口？各在什么情况下使用？体现为几种形式？具体为何？
7. 存储管理的 4 种主要功能是什么？
8. 简述操作系统的 3 种基本类型。

9. 什么是批处理系统? 什么是联机批处理和脱机批处理?

10. 什么是多道程序设计? 多道程序设计的目的是什么?

11. 为了更好地实现多道程序设计,你认为扩大内存容量和提高 CPU 速度哪个更重要?

12. 多道批处理系统的主要缺点是什么?

13. 简述分时操作系统的概念。

14. 实时操作系统的主要特征是什么? 主要应用在什么领域?

15. 简述实时操作系统的概念及特点。

16. 什么是网络操作系统? 它与个人计算机操作系统间的主要区别是什么?

17. 什么是分布式操作系统? 它与网络操作系统有什么区别?

18. 操作系统的 4 种基本特性是什么?

19. 如果没有并发特性,那么操作系统还会有其他 3 种基本特性吗?

20. 什么是并发和并行? 两者有什么区别?

21. 请说明操作系统中资源共享的两种方式。

22. 如何理解现代操作系统的虚拟特征?

23. 如何理解现代操作系统的异步特征?

24. 简述操作系统微内核结构的基本含义和意义。

25. 简述研究操作系统的几种观点。

26. 说一说自己常用的操作系统的特点。

第2章　用户接口

由操作系统的功能可知,操作系统不但是系统资源的管理者,而且要为用户使用计算机系统提供手段,这种手段就是用户使用计算机系统的接口,简称用户接口。该接口通常是以命令或系统调用的形式呈现在用户面前。前者通过键盘、终端或鼠标等方式提供给用户,具体有两种形式:命令方式(通过键盘、终端输入命令的使用方式,也称为命令接口)和图形方式(通过鼠标点击图标输入命令的使用方式,也称为图形接口);后者则提供给用户在编程时使用,也称为编程接口。

用户接口形式的变化也是体现操作系统不断发展的一个重要方面,下面就按照操作系统的发展来介绍用户接口的演变,即从脱机提交作业、联机命令交互到方便、直观、简单、易用的图形用户接口(Graphics User Interface,GUI)。

2.1　作　　业

2.1.1　作业的概念

1. 作业和作业步

作业是操作系统特别是具有批处理功能的操作系统中一个常见的概念。它可以从系统和用户两个方面进行解释。从系统角度可以按作业的组织形式定义作业,从用户角度可以从逻辑上抽象地(并非精确地)描述作业的定义。

先从用户角度对作业进行说明。用户为解决某个特定任务,先对它进行数学抽象,确定相应的数据结构和算法,然后用高级语言或者汇编语言进行程序设计。这种用高级语言或者汇编语言编写的程序就是源程序,再通过编译或汇编、连接、装配、运行等一系列步骤,计算机送出用户所需要的计算结果,这一过程如图2.1所示。把用户交由计算机进行加工处理的任务称为作业。在这一过程中,计算机系统要根据用户要求,执行一系列的工作,才能完成一个作业的运行。从问题提出、源程序提交给计算机系统到得出运算结果所经过的若干个加工步骤称为作业步。例如,计算机系统对用户的源程序可能要经过编辑(输入源程序和对程序进行修改)、编译或汇编(对用户程序进行语法检查和生成目标代码)、连接(将目标程序转换为可执行文件)、装入(将执行文件装入内存)、运行(系统启动运行目标程序得出计算结果)5步。每步(作业步)完成一项相对独立的工作。运行一个作业要经由一系列有序的作业步来完成,这些作业步相互关联,并且顺序地运行。一般而言,往往是上一个作业步的运行结果是下一作业步的输入信息。

从系统的角度看作业则是一个比程序更广的概念。它由程序、数据和作业说明书组成,系统通过作业说明书控制文件形式的程序和数据,使之执行和操作。而且,在批处理系统中作业是抢占内存的基本单位。也就是说,批处理系统以作业为单位把程序和数据调入内存以便执行。

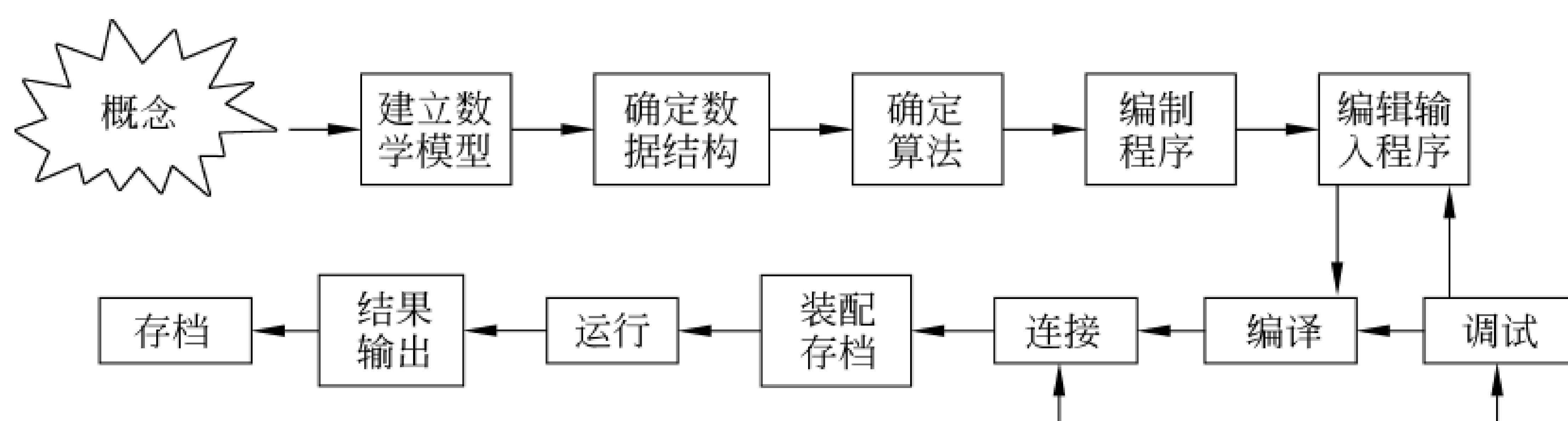


图 2.1 用户利用计算机解决某任务的过程

需要说明的是,作业的概念一般用于早期批处理系统和现在的大型机、巨型机系统中,对于广为流行的微机和工作站系统,人们一般不太使用作业的概念。

2. 作业的分类

根据计算机系统对作业处理方式不同,作业通常被分成两大类:批处理类型作业和交互式类型作业。

在批处理方式下,计算机系统一次可以成批接收多个用户作业,通过假脱机技术(如SPOOLing)将其放入磁盘的输入井(有关概念见第 7 章)中,然后等待操作系统中的作业调度程序进行调度。一旦被调度装入内存,才能被运行处理。批处理作业在输入计算机系统前,必须由用户使用作业控制语言编写一个作业说明书,作业控制程序解释作业说明书中的语句,根据作业的要求为其建立进程,从而控制作业运行。作业控制程序的工作过程如图 2.2 所示。

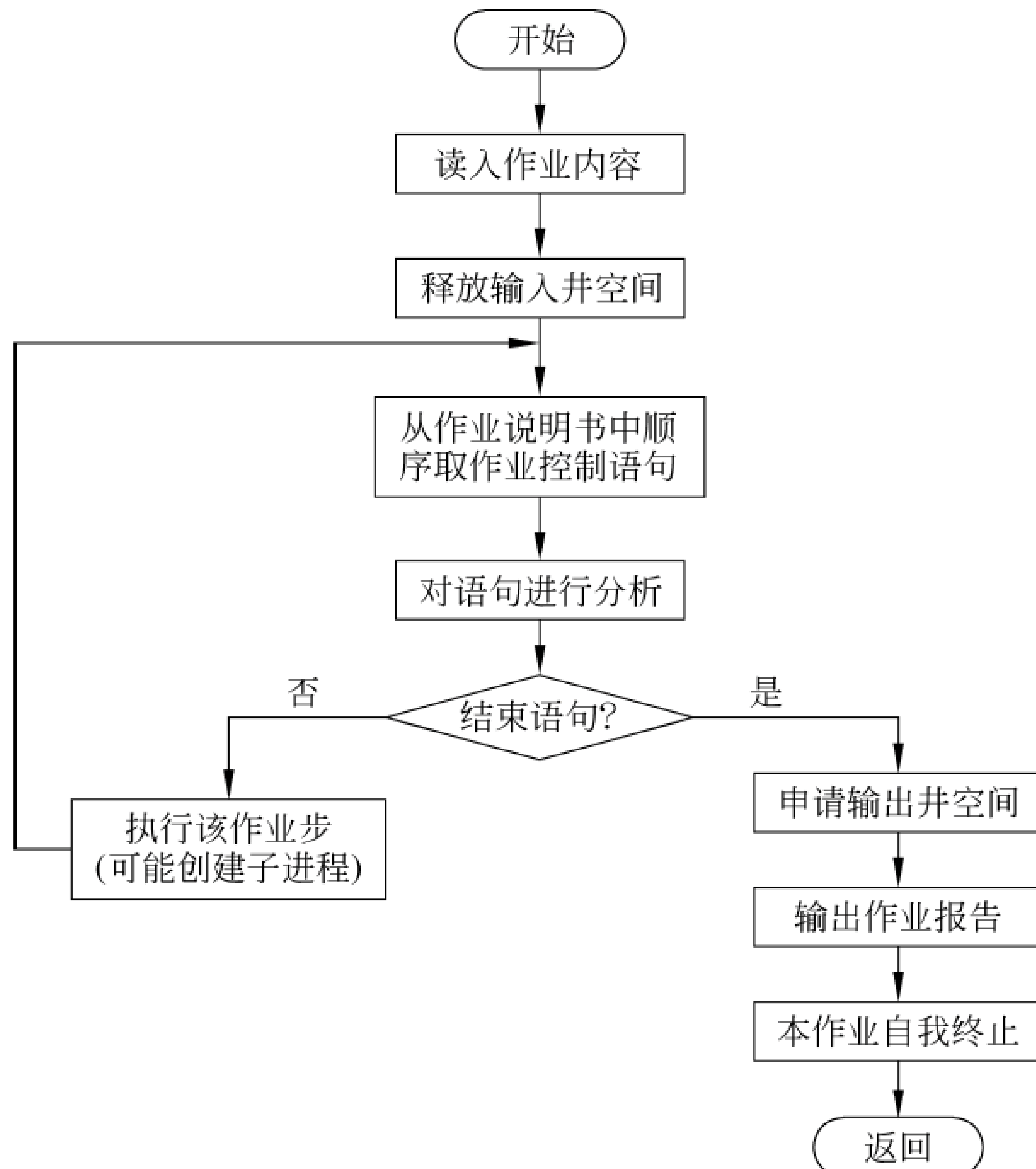


图 2.2 作业控制程序的工作过程

交互式作业又称终端作业。在分时系统中,用户在各自的终端上输入自己的用交互式会话语言(如 Basic)编制的源程序及有关数据,并且通过键盘、终端同计算机系统进行相互通信联系,告诉操作系统如何控制作业运行。操作系统也通过终端向用户报告运行情况和结果。

在同时具有分时操作和批处理两种方式的系统中,终端作业又称"前台"作业,而批处理作业又称为"后台"作业。终端作业要求及时响应,所以在作业调度时,其优先级(表示要求计算机处理的紧迫程度)要高于批处理作业。在终端作业负载轻时,才去调度批处理作业,以提高系统利用率。

2.1.2 作业控制块

作业控制块(Job Control Block,JCB)是作业在计算机系统中存在的唯一标识。作业控制块主要用来记录作业的标识、现行状态、作业的优先级和作业要求的资源等一系列关于作业运行的信息,它是作业运行的依据。在作业由提交状态转为收容状态时,由系统为这个作业创建一个作业控制块。为了管理方便,可将作业控制块通过链接技术连接成一条作业控制块队列,也可将不同状态的作业控制块组成不同的作业控制块队列。

作业控制块主要包括如下内容:

(1) 作业本身的内容,如作业的名字、程序作者名字和创建时间等。

(2) 为实现作业调度所需的信息,如作业本身的优先数(用数据表示优先级)、现在所处的状态以及所需使用中央处理机的时间等。

(3) 作业使用的资源要求,如该作业所需主存的大小、打印机和磁带机等。

(4) 系统指示单元,如该作业在辅存中的始址和长度等信息。

(5) 作业控制块的链接字,用以形成作业控制块队列。

2.1.3 作业的状态及其转换

1. 作业的状态

一个作业从进入计算机系统到运行结束一般可以分成提交、收容、执行和完成 4 个状态。

1) 提交状态

提交状态是指作业还未进入计算机系统之前,用户向系统提交作业过程时作业所处的状态。如在分时系统中,用户在各自的终端上编辑自己的程序(并未存盘),这个过程可称为提交状态。

2) 收容状态

收容状态是指用户提交的作业已通过脱机技术或 SPOOL 技术输入到辅存内所处的状态。收容状态又称后备状态。作业进入辅存后,系统为其建立作业控制块,并将其编入后备作业队列中,等待作业调度程序调度。

3) 执行状态

执行状态是指作业调度程序按一定的调度策略选中一个后备状态的作业,为其分配所需要的各种资源,从而使该作业具备了运行条件时所处的状态。这时有两种可能:一是该作业所需资源具备,因而一次即获得了全部所需资源,从而在处理机上运行(或等待处理

机);二是该作业所需的资源正在从事其他活动,该作业须等待其他活动完成。但总的来说,作业已经可以被看作正在执行了,因而把它此时所处的状态称为执行状态。

4)完成状态

作业正常运行结束或由于发生错误而终止运行时的状态称为完成状态。此时,系统将输出结果,收回所占资源,从现行队列中删除作业控制块。

2. 作业状态的转换

作业状态的转换过程如图2.3所示。

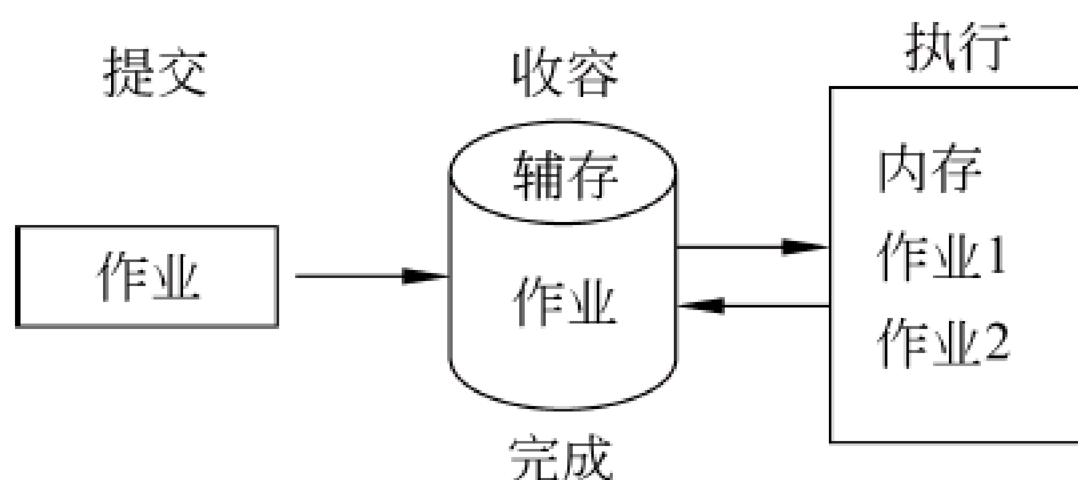


图2.3 作业状态转换图

作业从提交状态进入收容状态后,系统为其建立作业控制块,将其加入后备作业队列中,等待作业调度程序调度。当作业被作业调度程序按某种调度方法选中,且分配了必要的资源时,该作业便进入了执行状态。这时,处理机管理程序建立此作业的有关进程,转入进程调度。

作业运行结束后,由执行状态进入完成状态。系统通过作业调度程序将其运行结果(含出错信息)输出,回收其占用的系统资源,收回该作业控制块,并将其从现行作业队列中删除。系统可以通过联机方式,调用相应设备进程直接输出,也可使用SPOOL技术将其输出到输出井中,再调用相应的输出进程,将该作业的输出文件在输出设备上输出。

2.1.4 作业的输入输出方式

作业的输入是指通过作业输入设备把作业从输入介质送入系统(即磁盘的输入井)中并加以组织,在磁盘上建立一个后备作业的过程。作业的输出是指通过作业输处设备把系统(即磁盘的输出井)中作业的执行结果输出到一定介质上的过程。作业的输入输出方式主要有以下5种。

1. 联机输入输出方式

在联机输入输出方式中,主机与外围设备直接相连,一台主机可以控制一台或多台外围设备。外围设备在主机的直接控制下完成作业的输入输出。这样的外围设备可以是键盘、鼠标、显示器和打印机等。在这种方式中,由于主机和外围设备的速度相差悬殊,因而CPU的利用率较低。这种方式通常用在交互式系统中。

2. 脱机输入输出方式

在脱机输入输出方式中,主机与外围设备不直接相连,而是利用如个人计算机这样的低档计算机作为外围处理机进行输入和输出。在进行输入时,通过外围处理机把作业输入到后援存储器上,然后由人把存有作业的后援存储器挂接到与主机相连的外围设备上和主机连接。同样,在进行输出时,由人把存有作业运行结果的后援存储器从主机摘下,再连接到外围处理机上,外围处理机控制输出。在这种方式中,由于主机不用控制低速的外围设备进行输入输出,从而CPU的利用率得以提高。

3. SPOOL系统

SPOOL(Simultaneous Peripheral Operations On Line)可直译为外围设备同时联机操作。在SPOOL系统中,通过通道或DMA部件将多台外围设备与主机和外存连接在一起,作业的输入输出过程由主机中的操作系统控制完成。这种方式详见第7章。

4. 直接耦合方式

在直接耦合方式中，通过一个大容量的公用存储器把主机和多台外围处理机固定连接在一起，如图2.4所示。在这里外围处理机可由低档的个人计算机来担当。利用外围处理机把作业输入到公用存储器，再由主机进行处理，处理结果送到公用存储器中存储，待一个作业的结果全部存到公用存储器之后，再由外围处理机进行输出。

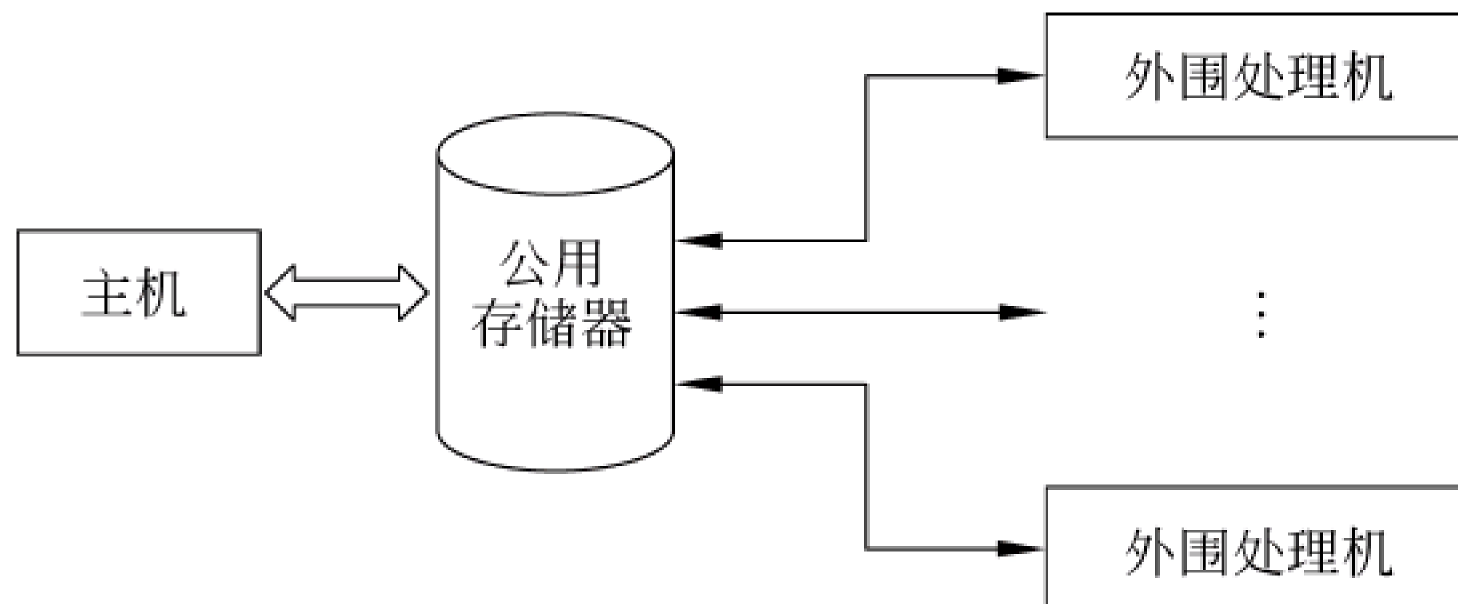


图2.4 直接耦合方式示意图

这种方式克服了联机输入输出方式CUP利用率低的缺点，同时克服了脱机输入输出方式中需要人工操作的缺点。

5. 网络输入输出方式

网络输入输出方式是以上述4种方式为基础的。利用上述4种方式进行作业输入之后，再通过网络把作业从存有作业的计算机中传送到处理作业的计算机中，作业处理完之后，再利用网络把结果传送到用来进行输出的计算机中进行输出。

2.2 命令接口

为了便于用户直接或间接地控制自己的作业，操作系统向用户提供了命令接口。用户可通过该接口向计算机发出命令以控制作业的运行。该接口又可进一步分为联机用户接口和脱机用户接口。

在分时系统和个人计算机中，操作系统向用户提供了一组联机命令，用户可以通过键盘终端输入命令，以取得操作系统的服务，并控制自己作业的运行。在批处理系统中，用户一旦把作业提交给系统后，便失去了自己直接与作业交互的能力，只能利用作业控制（命令）语言（Job Control Language，JCL）编写成作业说明书提交给系统后，由系统按用户作业说明书中的命令逐条解释执行。把分时系统中的接口称为联机命令接口，而把批处理系统中的接口称为脱机命令接口。

2.2.1 联机用户接口

联机用户接口是为联机用户提供的，它由一组键盘操作命令及命令解释程序组成。当用户在终端或控制台上每输入一条命令后，系统便立即转入命令解释程序，对该命令进行解释并执行该命令。在完成指定功能后，又返回到终端或控制台上，等待用户输入下一条命令。这样，用户可通过先后输入不同的命令来实现对作业的控制，直至作业完成。

为了使用联机命令接口，以实现用户与机器的交互，用户可通过键盘输入需要的命令，由终端处理程序接收该命令，并将它显示在终端屏幕上。当一条命令输入完后，由命令解释

程序对命令进行分析，然后执行相应命令的处理程序。可见，联机命令接口应包含命令、终端处理程序和命令解释程序。MS-DOS操作系统中，命令解释程序为COMMAND.COM，在Linux系统中为shell。

1. 联机命令格式

用户输入的命令通常以命令名开始，命令名本身标志着所要执行的操作。换言之，大多数命令都是通过运行某一特定程序来完成用户请求的操作的。此外，用户输入一条命令时，常常还须提供若干个参数，以指明一些辅助操作。在命令名和各参数之间须用分隔符（逗号、空格或分号等）分开，参数后还可带有某些用方括号括起来的可选项。命令的一般格式是：

```
Command arg1 arg2 … argn[option1 option2 … optionm]
```

2. 联机命令的类型

为了能向用户提供多方面的服务，通常，操作系统都向用户提供了几十条甚至上百条的联机命令，这些命令大致可分为以下几类：

(1) 环境设置。这些命令用来改变终端用户的所在位置和执行路径等。

(2) 执行权限管理。这些命令用来控制用户访问系统和读、写、执行有关文件的权限等，用户只有在其口令经过系统核准之后才能进入系统。

(3) 系统管理。该类命令主要用于系统维护、开机与关机、增加或减少终端用户、计时收费等。该类命令是操作系统提供的最为丰富的一类命令，且其中的很大一部分为系统管理员使用。

(4) 文件管理。该类命令被用来管理和控制终端用户的文件。例如，复制、移动或删除某个文件或显示文件内容和改变文件名字，以及搜索文件中的特定行或字符等。

(5) 编辑、编译、连接装配和执行。编辑命令被用来帮助用户输入用户文件，不同的编辑器具有不同的命令集合。这些命令被用来增加、删除输入字符或字符行，也被用来进行插入、移动甚至绘图等。编译和连接装配命令则把用户输入的源程序文件编译成目标代码文件之后再连接成可执行代码文件。执行命令则将连接后的可执行代码文件送入内存启动执行。

(6) 通信。通信类命令在单机系统中被用来进行主机和远程终端之间的呼叫、连接以及在主机和终端之间建立会话信道。在网络系统中，通信命令除了被用来进行有关信道的呼叫、连接和断开等之外，还进行主机和主机之间的信息发送与接收、显示和编辑等工作。

(7) 资源要求。用户使用该类命令向系统申请资源，例如申请某台外部设备等。

联机控制方式使用户直接参与控制作业执行，因而大大地方便了用户。但是在某些情况下，用户反复输入众多的命令也会感到非常烦琐或浪费了许多不必要的时间。例如，在对某个源代码文件进行编译调试之后，需要重新和多个目标代码文件连接。如果这个调试和连接不是一次成功的话（很多情况下是不可能一次成功的），那么用户的控制过程将会非常单调和烦琐。显然，在这种情况下，批处理方式要优于联机控制方式。因此，在现代操作系统中，大都提供批处理方式和联机控制方式。这里，批处理方式既指传统的作业控制语言编写的作业说明书方式，也指那些把不同的交互命令按一定格式组合后的命令文件方式。

2.2.2 脱机用户接口

脱机用户接口是为批处理作业的用户提供的,故也称为批处理用户接口。它由一组作业控制语言(JCL)的语句组成。批处理作业的用户不能直接与自己的作业交互作用,只能委托系统代替用户对作业进行控制和干预。JCL 便是提供给批处理作业用户的、为实现所需功能委托系统代为控制的一种语言。用户用 JCL 把需要对作业进行的控制和干预事先写在作业说明书上,然后将程序、数据和作业说明书一起提供给系统。当系统调度到该作业运行时,又调用命令解释程序,对作业说明书中的命令逐条地解释执行。如果作业在执行过程中出现异常现象,系统也将根据作业说明书中的指示进行干预。这样,作业一直在作业说明书的控制下运行,直至遇到作业结束语句时,系统才停止该作业的运行。

MS-DOS 操作系统中,通过编辑工具,按照用户的要求形成一个命令序列,以.bat 作为扩展名保存起来构成一个文件。这个文件就可作为作业的作业说明书,控制作业中的程序执行和数据的处理。在 Linux 系统中的 shell 也是一种程序语言,用来编写作业说明书。

2.3 编 程 接 口

编程接口是为用户程序在执行中访问系统资源而设置的,是用户程序获得操作系统服务的唯一途径。它由一组系统调用组成,每一个系统调用都对应一个能完成特定功能的子程序。如早期的 UNIX 系统版本和 MS-DOS 版本,它们的系统调用都是用汇编语言编写完成的,因而只有在用汇编语言编写的程序中才能直接使用系统调用,而在高级语言(如 C 语言)中,往往提供了与各系统调用对应的库函数,因而应用程序便可通过调用对应的库函数来使用系统调用。在如 UNIX System Ⅴ、Linux 和 OS/2 2.x 等版本的操作系统中,其系统调用是采用 C 语言编写的,并以函数形式提供,故在用 C 语言编制的程序中可直接使用系统调用。

系统调用是操作系统提供给编程人员的唯一接口。编程人员利用系统调用,在源程序一级动态请求和释放系统资源,调用系统中已有的系统功能来完成那些与机器的硬件部分相关的工作以及控制程序的执行等。因此,系统调用像一个黑箱子那样,对用户屏蔽了操作系统的具体实现细节而只提供有关的功能。事实上,命令接口也是在系统调用的基础上开发而成的。

系统调用是中断的一种,是自愿性中断,关于中断在第 7 章中详细介绍,本节只介绍系统调用作为编程接口的有关内容。

2.3.1 系统调用的类型

不同的操作系统提供了不同的系统调用。一般一个系统为用户提供几十到几百条系统调用。系统调用大致可分为如下几类:

(1) 设备管理。该类系统调用被用来请求和释放有关设备以及启动设备等操作。

(2) 文件管理。包括对文件的读、写、创建和删除等。

(3) 进程控制。进程是一个在功能上独立的程序的一次执行过程。进程控制的有关系统调用包括进程创建、进程执行、进程撤销、进程等待和进程优先级控制等。

（4）进程通信。该类系统调用被用在进程之间传递消息或信号。

（5）存储管理。包括查询作业占据内存区的大小、获取作业占据内存区的始址等。

（6）线程管理。包括线程的创建、调度、执行和撤销等。

进程和线程的概念以及存储管理、文件管理和设备管理的具体内容将在后续章节中详细介绍。

2.3.2 系统调用的实现

为了提供系统调用功能，操作系统内必须有事先编制好的实现这些功能的子程序、过程或函数。显然，这些子程序或过程是操作系统程序的一部分，且不能直接被用户程序调用。而且，为了保证操作系统程序不被用户程序破坏，一般操作系统都不允许用户程序直接访问操作系统的系统程序和数据。那么，编程人员给定了系统调用名称和参数之后是怎样得到系统服务的呢？这需要有一个类似于硬件中断处理的中断处理机构（详见第 7 章），当用户使用系统调用时，产生一条相应的指令，处理机在执行到该指令时发生相应的中断，并输出有关信号给该处理机构，该处理机构在收到了处理机发来的信号后，启动相关的处理程序去完成该系统调用所要求的功能。

将系统中控制系统调用的服务机构称为陷阱（trap）处理机构。与此相对应，把由于系统调用引起处理机中断的指令称为陷阱指令（或称访管指令）。在操作系统中，每个系统调用都对应一个事先给定的功能号，例如 0，1，2，3 等。在陷阱指令中必须包括对应系统调用的功能号。而且，在有些陷阱指令中，还带有传递给陷阱处理机构内部处理程序的有关参数。

为了实现系统调用，系统设计人员还必须为实现各种系统调用功能的子程序编造入口地址表，每个入口地址都与相应的系统子程序名对应起来。然后，由陷阱处理程序把陷阱指令中所包含的功能号与该入口地址表中的有关项对应起来，从而由系统调用功能号驱动有关系统子程序执行。

由于在系统调用处理结束之后，用户程序还需利用系统调用的返回结果继续执行，因此，在进入系统调用处理之前，陷阱处理机构还需保存处理机现场。在系统调用处理结束之后，陷阱处理机构还要恢复处理机现场。在操作系统中，处理机的现场一般被保护在特定的内存区或寄存器中。

有关系统调用的另一个问题是参数传递问题。不同的系统调用需要传递给系统子程序不同的参数。而且，系统调用的执行结果也要以参数形式返回给用户程序。实现用户程序和系统程序之间参数传递的方法有如下 3 种：

（1）由陷阱指令自带参数。一般来说，一条陷阱指令的长度总是有限的。而且，该指令还要携带一个系统调用的功能号，故陷阱指令只能自带极少的几个参数进入系统内部。

（2）通过使用有关通用寄存器来传递参数。显然，这些寄存器应是系统程序和用户程序都能访问的。不过，由于寄存器个数有限，无法传递较多的参数。

（3）通过堆栈区来传递参数。在系统调用带有较多的参数时使用该方法。

另外，在系统发生访管中断或陷阱中断时，为了不禁止用户程序直接访问系统程序，反映处理机硬件状态的处理机状态字（PSW）中的相应位要从用户执行模式转换为系统执行模式。这一转换在发生访管中断或陷阱中断时由硬件自动实现。一般把处理机在用户程序

中的执行状态称为用户态，而把处理机在系统程序中的执行状态称为核心态（或称系统态）。

2.4 图形接口

用户虽然可以通过联机用户接口来取得操作系统的服务，并控制自己的程序运行，但要求用户记住各种命令的名字和格式，并严格按照规定的格式输入命令，既不方便又花费时间。于是，图形用户接口便应运而生。

图形用户接口采用了图形化的操作界面，用非常容易识别的各种图标（icon）将系统的各项功能、各种应用程序和文件直观、逼真地表示出来。用户可通过鼠标、菜单和对话框来完成对程序和文件的操作。此时用户已完全不必像使用命令接口那样去记住命令名及格式，从而把用户从烦琐且单调的操作中解脱出来，也使计算机成为一种非常有效且生动有趣的工具。

图形用户接口可以方便地将文字、图形和图像集成在一个文件中。可在文字型文件中加入一幅或多幅彩色图画，也可以在图画中写入必要的文字，而且还可进一步将图画、文字和声音集成在一起。

20 世纪 90 年代推出的主流操作系统都提供了图形用户接口。例如，IBM 公司的 OS/2 2.1 OS、Apple 公司的 Macintosh OS、Microsoft 公司的 Windows 和 Linux。

图形用户接口的元素主要有窗口、图标、菜单和对话框。

2.4.1 窗口

1. 窗口的组成

窗口的组成如图 2.5 所示。不同应用程序或文档的窗口的组成可能不同，但有些元素是各类窗口所共有的。



图 2.5　窗口的组成及元素

窗口所共有的元素主要有：

（1）控制菜单图标；

（2）标题拦和窗口标题；

（3）最大化按钮、最小化按钮、还原按钮和关闭按钮；

（4）菜单栏和菜单；

（5）水平标尺和垂直标尺；

（6）水平滚动框和垂直滚动框；

（7）编辑区、鼠标指针和插入点；

（8）状态栏等。

2. 窗口分类

可以从不同角度对窗口进行分类。

（1）从系统的角度进行分类：系统窗口和用户窗口。

（2）按应用程序分类：应用程序窗口和包含在应用程序窗口中的窗口。

3. 窗口的性质

窗口是具有一定性质的矩形区域，其性质如下：

（1）窗口的状态：打开状态和激活状态。

（2）窗口的改变：窗口大小的改变、窗口位置的改变。

4. 窗口的操作

对窗口的主要操作如下：

（1）移动窗口。

（2）改变窗口大小。

（3）滚动条的使用。

（4）关闭窗口。

2.4.2 图标

所谓图标是代表一个应用程序或文件等的一个小图像，也是最小化的窗口，通过图标的操作可以激活相应的程序或启动应用程序。图标主要有 3 类：

（1）应用程序图标；

（2）文件夹；

（3）应用程序项图标。

对图标可以进行移动，可以把窗口收缩为图标等。

2.4.3 菜单

菜单由菜单名和若干菜单项组成。每个菜单项通常都对应一个相关的命令或功能，有时也可以是打开的窗口、文件等列表，或是图形或文件特征的表项。用户可以通过鼠标或键盘在菜单中选择一个菜单项，向系统提出相应的服务请求。

菜单的形式有菜单条、弹出式菜单和下拉式菜单。

菜单的类型主要有应用程序菜单、窗口控制菜单和快捷菜单。

对菜单的主要操作有选择和关闭应用程序菜单、选择菜单命令等。

2.4.4 对话框

对话框是在桌面上的带有标题和控制菜单的一个临时窗口，也称为对话窗口。其主要用途是：系统可通过对话框提示用户输入与任务有关的信息，或向用户提供可能需要的信

息。在菜单中所选择的命令后面带有省略号(…)时，表示选择该命令后将立即出现一个对话框。例如，当用户在Word(Windows系统提供的一个字处理软件)的文件菜单中，选择了"打开"命令(其后面有省略号(…))后，将出现对话框，提示用户输入要打开的文件名、其所在目录、所在驱动器及文件类型等信息，如图2.6所示。在用户根据对话框中的提示输入相关信息后，指定的文件即被打开，对话框消失。



图2.6 对话框的形式

对对话框的主要操作如下：

(1) 选择对话框。经常可以在一个对话框中又出现多个小对话框或命令按钮供用户选择或应答，但一次只能有一个小对话框或按钮被激活。所以，当需要对其中的某个小对话框进行操作时，应该先选择(激活)它，被选中的小对话框可呈现出某种特性，如高亮或用虚线勾出轮廓等。可以用鼠标或键盘来激活小对话框。

(2) 文本框的操作。文本框是在对话框中所出现的、供给用户输入文本信息的小对话框。对文本框的操作包括文本的输入、在文本内移动、对输入错误进行修改和对要修改的文件进行选择等。

(3) 列表框的操作。列表框显示多个可选的列表项。当其窗口大小不足以显示全部列表项时，就将出现滚动条。用户可从中选出一个选项。有时也可选出多个选项。

(4) 按钮操作。在对话框中常出现的按钮有命令按钮、单选按钮和复选框。

2.5 Linux的用户接口

Linux系统给用户提供了方便的使用界面，它的用户接口形式有命令接口、图形接口和编程接口。

2.5.1 Linux命令接口

在Linux中的命令shell程序来解释执行的。shell是Linux系统的最外层，也称为外壳。它可以作为命令语言为用户提供使用操作系统的接口，同时也是一种程序设计语言，用户可以利用多条shell命令构成一个文件。

Linux 有数以百计的命令,在这里不进行详细介绍,只做概括说明,详见 Linux 的命令手册。

Linux 的命令主要有如下几类。

1. 系统设置类

alias:设置指令的别名。alias 的作用只限于本次登入的操作。

clock:调整 RTC(计算机内建的硬件时间)时间。

passwd:用来更改使用者的密码。

2. 系统管理类

exit:使 shell 以指定的状态值退出。

login:让用户登录系统,亦可通过它的功能随时更换登录身份。

logout:让用户退出系统,其功能和 login 指令相互对应。

kill:删除执行中的程序或工作,同时可将指定的信息送至程序。

shutdown:用来进行关机,并且在关机以前传送消息给所有使用者正在执行的程序,也可以用来重开机。

who:显示系统中的使用者。

bg:将程序放在后台执行。

fg:将后台任务拉到前台执行。

clear:清除终端屏幕。

man:格式化和显示在线手册。

3. 文件管理类

chmod:Linux/UNIX 的文件调用权限分为 3 级:文件拥有者、群组和其他人。利用 chmod 可以控制文件如何被他人所调用。该命令中的参数为:u(设置目录的所有人),g(群组),o(其他人),r(读),w(写)和 x(执行)。

cp:将一个文件复制至另一文件,或将数个文件复制至另一目录。

find:将文件系统内符合指定要求的文件列出来。

ln:链接文件或目录。Linux/UNIX 文件系统中,有所谓的链接(link),可以将其视为文件的别名,而链接又可分为两种:硬链接(hard link)与软链接(symbolic link),硬链接的意思是一个文件可以有多个名称,而软链接的方式则是产生一个特殊的文件,该文件的内容是指向另一个文件的位置。硬链接是存在同一个文件系统中,而软链接却可以跨越不同的文件系统。

mv:将一个文件移至另一个文件,或将数个文件移至另一目录。

rm:删除文件及目录。

4. 备份压缩类

gzip:压缩程序,文件经它压缩后,其名称后面会多出.gz 的扩展名。

gunzip:用于解开被 gzip 压缩过的文件,这些压缩文件默认的扩展名为.gz。事实上 gunzip 就是 gzip 的硬链接,因此不论是压缩或解压缩,都可通过 gzip 命令单独完成。

tar:备份文件。

zip:压缩程序,文件经它压缩后会产生具有 zip 扩展名的压缩文件。

unzip:解压缩经 zip 压缩的文件。

5. 磁盘管理类

cd：变换工作目录。

df：显示文件系统的状况，或是查看所有文件系统的状况(默认值)。

ls：显示指定工作目录下的内容(列出目前工作目录所含的文件及子目录)。

mkdir：建立子目录。

mount：将某个文件的内容解读成文件系统，然后将其挂在目录的某个位置之上。

umount：卸除目前挂在 Linux 目录中的文件系统。

6. 磁盘维护类

dd：转换及输出数据命令。

fdisk：用于观察硬盘的实体使用情况和分割硬盘。

swapoff：该命令实际上是 swapon 的符号链接，可用来关闭系统的交换区。

Swapon：开启系统的交换区。

7. 网络通信类

ping：执行 ping 命令会使用 ICMP 传输协议，发出要求回应的信息，若远端主机的网络功能没有问题，就会回应该信息，因而得知该主机运作正常。

talk：与其他使用者对话。

telnet：开启终端机阶段作业，并登录远端主机。

write：传讯息给其他使用者。

8. 电子邮件新闻组类

mail：收发邮件。

pine：收发邮件。

9. 文件传输类

bye：在 FTP 模式下，输入 bye 即可中断目前的连线作业，并结束 FTP 的执行。

ftp：连接 FTP 服务器。

ncftp：传输文件。

10. 文本编辑类

ed：是 Linux 中功能最简单的文本编辑程序，一次仅能编辑一行而非全屏幕方式的操作。

grep：用于查找内容包含指定的范本样式的文件。

joe：是一个功能强大的全屏幕文本编辑程序。

pico：是一个简单易用、以显示导向为主的文字编辑程序，它随处理电子邮件和新闻组的程序 pine 一起提供。

vi：屏幕编辑程序。

11. 打印作业类

cat：把显示文件内容。

more：分屏显示文件内容。

less：其作用与 more 十分相似，都可以用来浏览文字文件的内容，不同的是 less 允许使用者往回卷动以浏览已经看过的部分。

cut：提取并显示特定列的内容。

tail：显示文件的末尾内容。

12. X Window System 类

包括 startx 启动图形界面等命令。

13. 格式转换类

giftopnm：将 GIF 格式图形文件转换为 PNM 格式。

pfttops：转换字体文件。

14. 特殊命令

≫：在命令后面跟此符号和文件名，则将命令的执行结果追加到该文件原有内容之后。

＞：在命令后面跟此符号和文件名，则将命令的执行结果输出到该文件中，该文件中原有的内容被删除。

＆：该符号与≫或＞联合使用，将命令的执行结果同时输出到标准输出设备上和文件中。

＜：在命令后面跟此符号和文件名，则在执行命令时从该文件中提取命令所需的输入数据。

上述 4 个命令也称为输入输出重定向命令。

2.5.2 Linux 编程接口

Linux 的编程接口也称为系统调用或 Linux C 函数，是供程序员在编程中使用的，它允许程序员不必了解系统程序的内部结构和与硬件有关的细节就可实现相应的功能，从而降低了程序员设计和编写程序的难度，保护了系统资源，提高了资源的利用率。

Linux 系统中包含 20 多个类的 400 多个常用函数。下面列举一些常用的函数，以便后续章节使用。

1. 用户管理类

getuid()：获取用户标识号。

geteuid()：获取有效用户标识号。

2. I/O 类

open()：打开文件。

close()：关闭文件。

read()：读文件。

write()：写文件。

sync()：将缓冲区数据写回磁盘。

3. 进程控制类

fork()：创建一个新进程。

exit()：正常结束进程。

execl()：运行可执行文件。

getpid()：取得进程识别码。

wait()：等待子进程中断或结束。

4. 进程通信类

kill()：传送信号给指定进程或进程组。
signal()：设置信号处理方式。
pause()：让进程暂停直到信号出现。
msgctl()：控制信息队列的运转。
msgget()：建立信息队列。
msgrev()：从信息队列读取信息。
msgsnd()：将信息送入信息队列。
pipe()：建立管道。

5. 存储管理类

malloc()：分配内存空间。
free()：释放原先分配的内存。
getpagesize()：取得内存页面大小。
mmap()：映射虚拟内存页。
munmap()：去除内存页映射。

6. 系统管理类

time()：取得系统时间。
times()：取进程运行时间。
getrusage()：获取系统资源使用情况。

7. 线程管理类

pthread_create()：创建一个线程函数。
pthread_exit()：线程通过调用该函数来终止自身执行。
pthread_join()：用来等待一个线程的结束。

2.5.3 Linux 的图形接口

图 2.7 是 Linux 系统的图形化桌面，桌面底部 GNOME 面板中的图标依次表示的操作为：

- 开始菜单；
- Mozilla(系统默认浏览器)；
- Evolution(系统默认邮件客户端系统)；
- OpenOffice.org Writer(字处理程序)；
- OpenOffice.org Cal(电子表格系统)；
- OpenOffice.org Impress(幻灯制作及演示文稿系统)；
- 打印；
- 工作区切换器；
- 通知区域；
- 日期和时间区域。

图 2.8 是 Linux 启动字处理程序后的一个窗体，图 2.9 是 Linux 启动电子表格处理程序后的一个窗体。

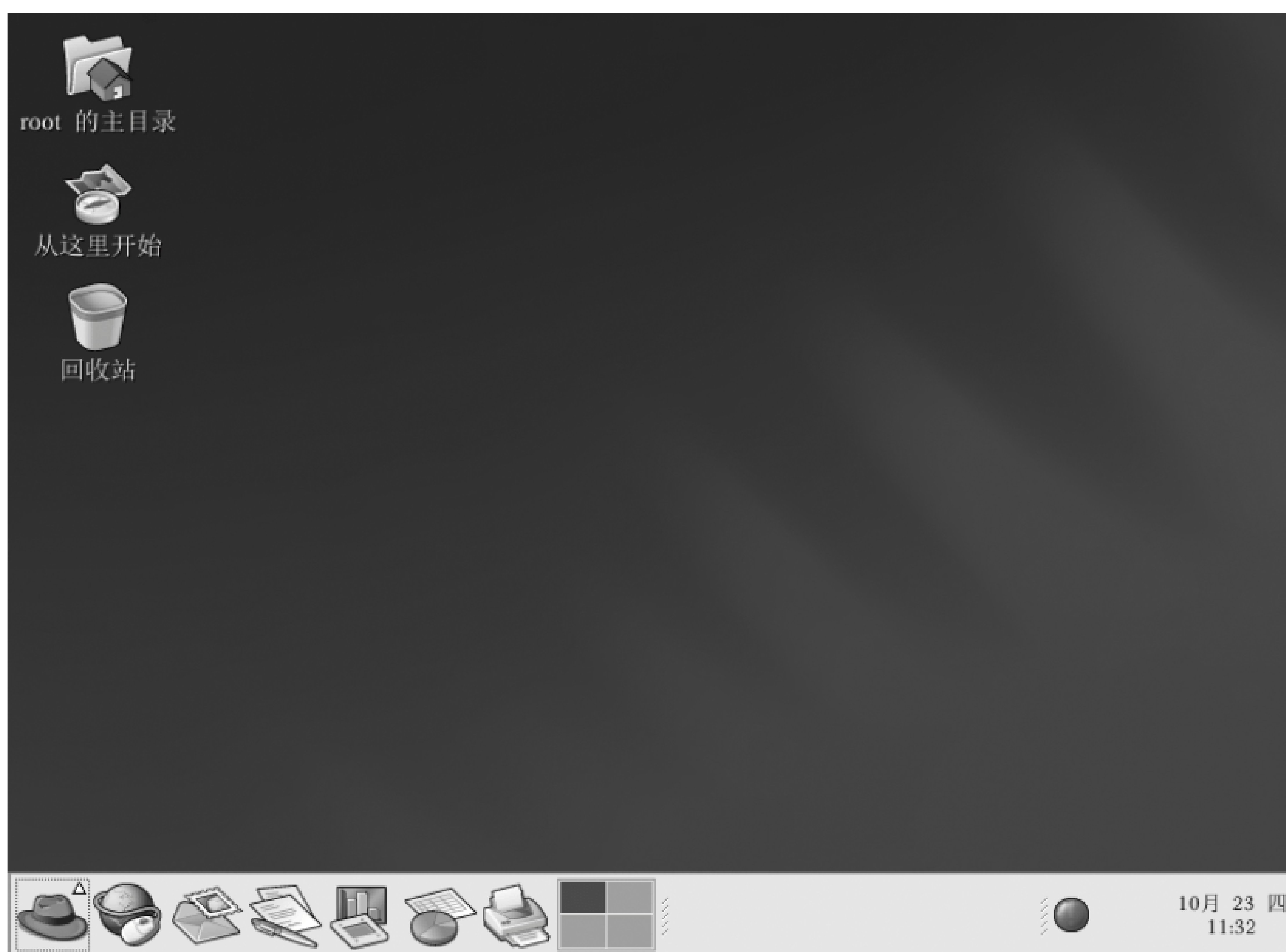


图 2.7　图形化桌面

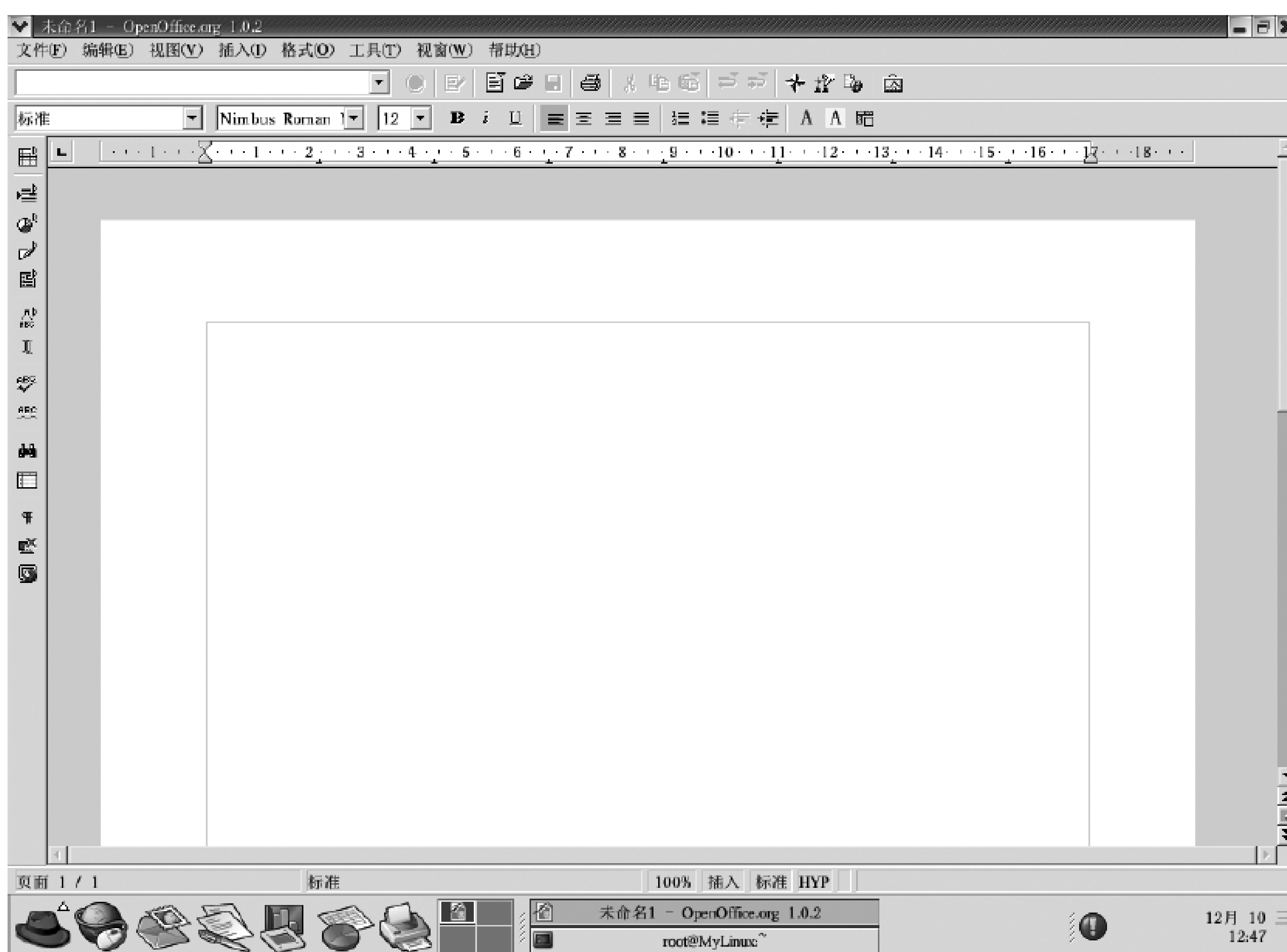


图 2.8　在 Linux 中启动字处理程序后的一个窗体

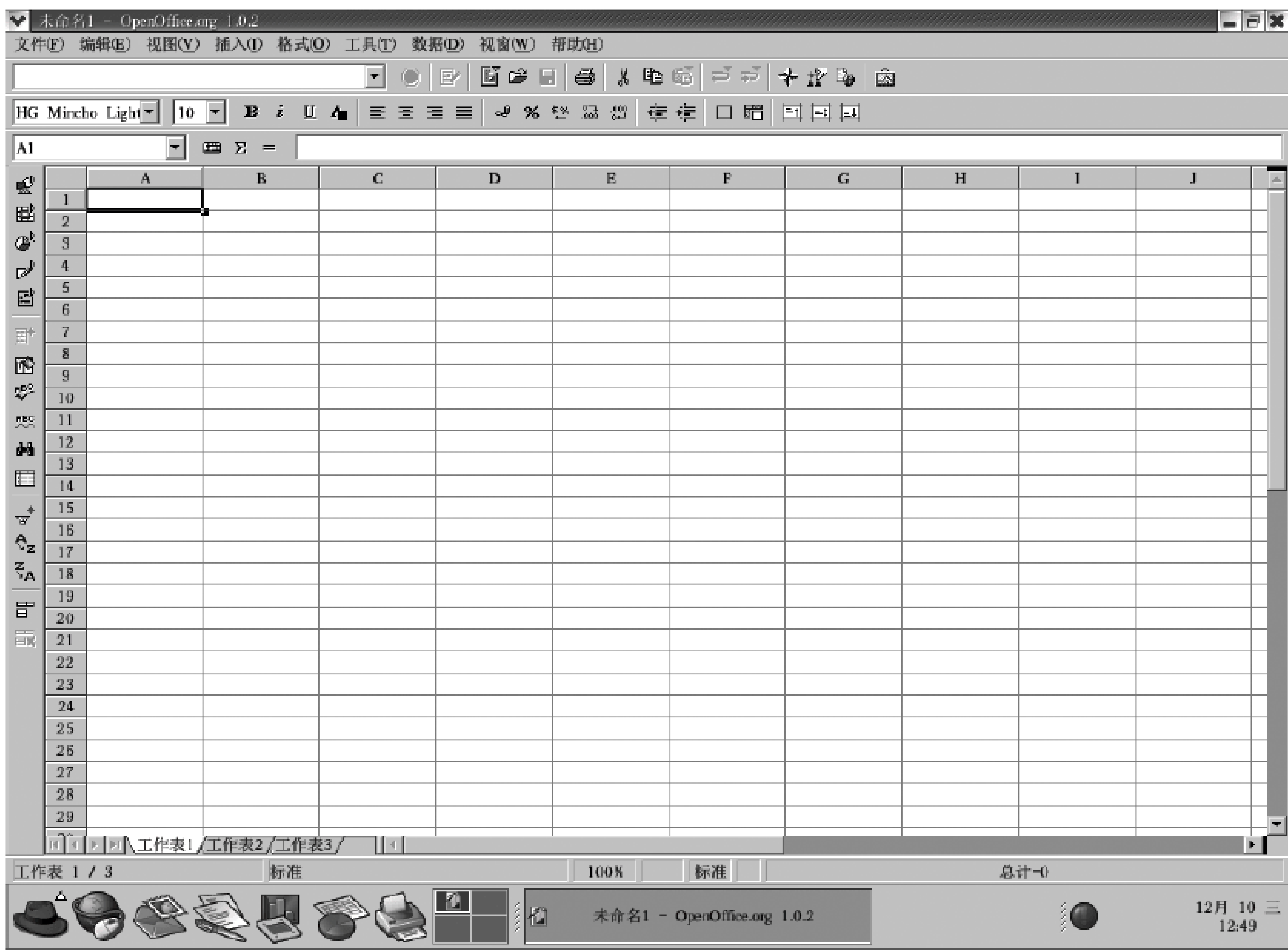


图 2.9　在 Linux 中启动电子表格处理程序后的一个窗体

2.6 本章小结

本章首先介绍了作业的相关内容：作业的概念、分类和标识，作业的状态及其转换，作业的输入输出方式。

随后介绍了操作系统的用户接口。操作系统接口是操作系统向用户提供功能的途径，是评价操作系统的一项重要指标。操作系统的用户接口有两种方式，具体体现为 3 种表现形式：命令接口、编程接口（即系统调用）和图形接口。命令接口和图形接口是建立在系统调用（编程接口）方式基础之上的；命令接口需要用户记住文本命令，但执行方式简洁；图形接口是把文本命令图形化，不需要用户记住命令，只需点击相应的图标即可；编程接口是操作系统提供给编程人员的唯一接口，编程人员通过系统调用使用操作系统所提供的各种功能。系统调用在核心态下执行。

Linux 操作系统为用户提供了以上 3 种接口，使用起来非常方便。本章对 Linux 及其 3 种接口进行了简单的介绍。

习　　题

1. 操作系统接口指的是什么？它有几种方式？
2. 举例说明作业和作业步。
3. 为什么说在分时系统中没有作业的概念？
4. 作业存在的标志是什么？它包含哪些信息？

5. 画图说明作业的状态及其转换。

6. 说明作业的各种输入输出方式。

7. 命令接口主要用在哪种场合? 主要分几类?

8. 编程接口又称为什么? 它用在何种场合?

9. 图形接口的优点是什么? 在一般的图形窗口中包含哪些元素? 常用的操作方式有哪几种?

10. Linux 系统为何更新得特别快?

11. 安装 Linux 系统,熟悉它的图形操作方式。

12. 对本章讲述的 Linux 命令进行使用。

第3章 进 程 管 理

早期的计算机系统一次仅允许执行一个程序。这个程序占有计算机系统的所有资源。现代计算机系统则允许多道程序同时装入内存并且可以同时执行。这种变化给现代操作系统的研究和设计提出了许多新问题,它要求操作系统对内存中的多个作业或程序进行有效的协调和控制,以保证它们互不影响,而且能够正确、可靠地执行。为了能够精确地描述和研究程序在系统中运行的动态情况,就引入了进程这一概念。进程可以认为是处于执行状态中的程序,也可以认为是现代操作系统中的一个基本运行单位。进程是操作系统中最重要的一个概念,它很好地刻画了系统内多道程序的动态性和并发特性,为程序的执行和系统资源的管理等提供了一种有效的描述手段。

3.1 进程的基本概念

在单道程序环境下,程序的执行是以顺序方式进行的,即必须一个程序执行完后下一个程序才能执行。在多道程序环境下,则允许多个程序并发执行,而程序并发执行的诸多特征很难用传统程序的概念来描述,故操作系统中引入了进程的概念。由此可见,进程与程序的动态执行和并发特性有很大关系,下面就先分析程序的顺序执行和并发执行的特点,以引出进程的概念。

3.1.1 程序的顺序与并发执行

1. 程序的顺序执行及其特征

通常,一个程序由若干个程序段组成,各程序段必须按照事先规定的次序顺序执行,只有在当前程序段执行完以后下一个程序段才能执行。例如,一般程序包括3个部分:输入部分Pi、计算部分Pc和输出部分Po。程序的执行过程就是输入一组数据,并对它们进行处理,然后输出计算结果,其执行过程如图3.1所示。显然,对于这样的程序执行方式,只有前一程序段执行完,后续程序段才能执行,故称这样的程序是顺序执行的。把一个具有独立功能的程序段独占处理机,依次执行其每条语句,直至得到最终结果的过程称为程序的顺序执行。

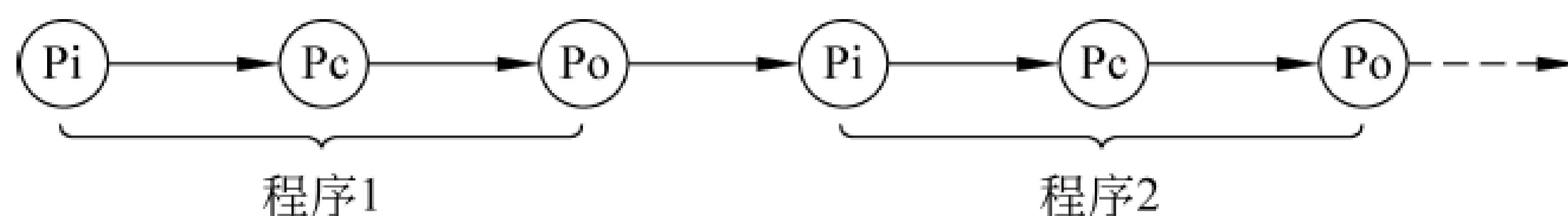


图3.1 程序的顺序执行过程

通过上面的分析可以看出,程序的顺序执行具有如下特征。

(1) 顺序性:每一操作必须在下一操作开始之前结束。

(2) 封闭性:程序在封闭的环境中运行,程序独占计算机系统的全部资源,程序的执行结果由给定的初始条件决定,不受外界因素的影响。

(3) 可再现性:程序的运行结果与执行速度无关,只要初始条件相同,任何时候执行都会得到相同的结果。

程序的顺序执行使得程序独占系统的所有资源,而程序在其执行的某一刻不可能同时使用系统的所有这些资源,故程序的顺序执行导致系统资源的严重浪费。

2. 多道程序执行环境及其特点

程序的顺序执行导致系统资源的浪费,此时系统内仅存在一道程序,某一时刻它仅占用系统的一部分资源,而没有被其使用的其他资源处于闲置状态。为了提高资源的利用率,引入了多道程序的概念,即内存中同时存在多个程序,分别占有或轮流使用系统中的不同资源,以达到提高系统资源利用率的目的。例如,系统中有 3 道程序,每个程序均由输入、计算和输出 3 部分组成,那么第一道程序输入结束进入计算阶段时,第二道程序就可以使用输入部件进行输入;当第一道程序计算结束进行输出时,第二道程序就可进入计算阶段,第三道程序就可使用输入部件进行输入,此过程如图 3.2 所示。由此可见,只要系统中存在一定数量的程序,这些程序的执行顺序得到好的安排,系统各部分资源就可以尽可能地忙起来,这样不但提高了系统资源的利用率,而且系统的处理能力也得到了增强。

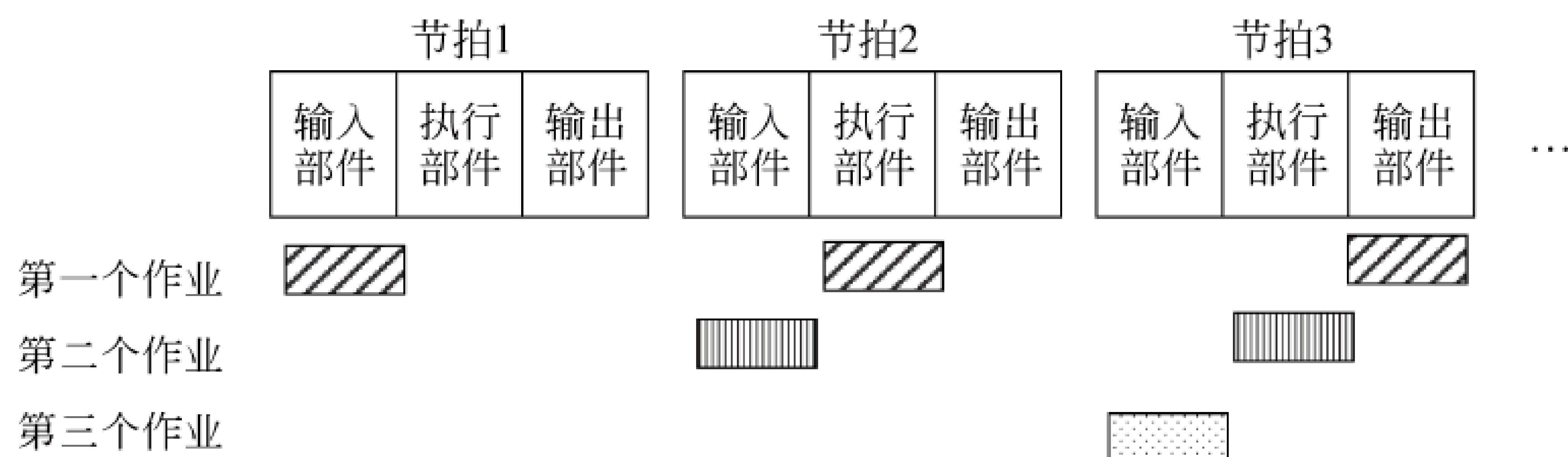


图 3.2 多道程序执行过程示意图

多道程序系统引起了程序执行环境的变化,这种执行环境一般称为多道程序执行环境。这种执行环境主要有以下 3 个特点:

(1) 独立性。在此执行环境中存在多道程序的执行,每道执行的程序在逻辑上都是独立的,各道执行的程序之间在逻辑上不存在制约关系。

(2) 随机性。在此执行环境中,特别是多用户环境,每道程序何时装入内存、加工处理的数据何时输入以及程序何时开始执行都是随机的,不是由用户指定的,而是根据系统中程序的运行情况和资源使用情况等决定。

(3) 资源共享性。系统拥有的各种资源由系统存在的各道程序共同使用,即系统资源是执行者的程序的共享资源。一般来讲,计算机内运行的程序道数多于系统所具有的 CPU 的数量,而且其他的系统资源也是有限的,这样必然导致各道程序对有限的系统资源的激烈竞争以及相互制约,多道程序导致了系统复杂性的急剧增加。

3. 程序的并发执行及其特征

先来看由语句 S1 至 S4 所组成的一段程序:

S_1: p=x+13

S_2: q=y-24

S_3: m=p+q-12

S_4: n=n+m+16

当顺序执行时，语句 S_1、S_2、S_3 和 S_4 依次执行。不难发现 S_1 和 S_2 是相互独立的两条语句，没有相互依赖关系，执行次序可以变换，即语句 S_1 和 S_2 谁先执行不影响执行结果的正确性，即语句 S_1 和 S_2 可以并发执行。该段程序的执行过程如图 3.3 所示。显然语句 S_1 和 S_2 的并发执行会加快该程序段的执行速度。程序内部的这种并发特性可以在编程时通过并行程序设计手段显式地描述出来。

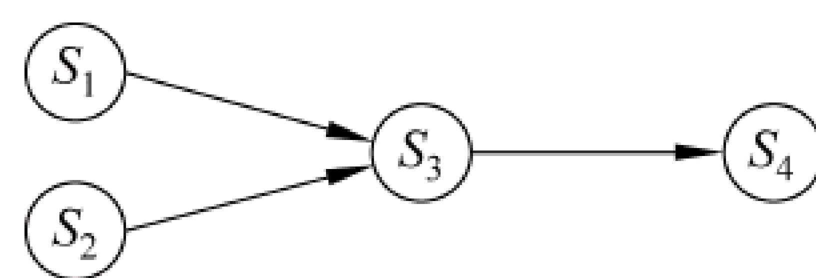


图 3.3　程序的并发执行

图 3.3 称为前趋图。前趋图是用来描述一个程序的各部分(程序段或语句)间的依赖关系，或者是一个大的计算的各个子任务间的因果关系。前趋图中的每个节点可以表示一条语句、一个程序段或一个进程，节点间的有向边表示两个节点之间存在的偏序(partial order)或前趋关系(precedence relation)。

设语句 S_1 和 S_2 是两条相邻的语句，它们可以并发执行的条件在 1966 年由 Bernstein 提出。将程序中的任何一条语句 S 划分为两个变量集 $R(S)$ 和 $W(S)$，$R(S)$ 是语句 S 在执行期间进行读的变量的集合，$W(S)$ 是语句 S 在执行期间进行写的变量的集合。

对于两条相邻的语句 S_1 和 S_2，如果下述 3 条同时成立，则语句 S_1 和 S_2 可以并发执行。

$$R(S_1) \cap W(S_2) = \phi$$
$$W(S_1) \cap R(S_2) = \phi$$
$$W(S_1) \cap W(S_2) = \phi$$

例 3.1　根据 Bernstein 条件，则有以下 4 条语句。

$$S_1: a = x + y$$
$$S_2: b = z + 1$$
$$S_3: c = a + b$$
$$S_4: w = c + 1$$

则 S_1 和 S_2 两条语句、S_3 和 S_4 两条语句是否可以并发执行？

解：这 4 条语句对应的读集和写集分别如下：

$$R(S_1) = \{x, y\}$$
$$R(S_2) = \{z\}$$
$$R(S_3) = \{a, b\}$$
$$R(S_4) = \{c\}$$
$$W(S_1) = \{a\}$$
$$W(S_2) = \{b\}$$
$$W(S_3) = \{c\}$$
$$W(S_4) = \{w\}$$

对于 S_1 和 S_2 两条语句，有

$$R(S_1) \cap W(S_2) = \phi,\quad R(S_2) \cap W(S_1) = \phi,\quad W(S_1) \cap W(S_2) = \phi$$

所以 S_1 和 S_2 语句可以并发执行。

对于 S_3 和 S_4 两条语句两条语句，有

$$R(S_3) \cap W(S_4) = \phi,\quad R(S_4) \cap W(S_3) \neq \phi,\quad W(S_3) \cap W(S_4) = \phi$$

所以 S_3 和 S_4 语句不可以并发执行。

与程序内语句间的并发特性相似，同样逻辑上相互独立、不存在相互依赖关系的多个程序段间也可能存在并发性，进而程序间也可能并发地执行。而在一定的硬件条件下（如多CPU系统），这种并发性可以转变为并行性，从而大幅度提高硬件资源的利用率，加快程序执行的速度。

由上面的分析可知，程序的并发执行是为了提高计算机系统的处理能力和提高资源利用率所采取的一种同时操作技术，使得一组在逻辑上相互独立的程序、程序段或程序语句在执行过程中，其执行时间在宏观上相互重叠。

程序的并行执行不同于程序的并发执行。程序的并行执行是指一组程序按独立的、异步的速度执行。并行执行不等于时间上的重叠。

显而易见，程序的并发执行使得系统资源尽可能地处于满负荷状态，提高了系统资源的利用率，从而显著增强了系统的处理能力。但因多道程序的并发执行及其对系统资源的共享和竞争而导致了系统复杂性的急剧增加，此时系统具有如下特征：

（1）程序执行的间断性。资源的有限性以及对资源的共享和竞争导致程序执行速度的改变，使得程序的执行具有间断性，程序间相互制约，使程序具有“执行—暂停—执行—暂停—…”的活动规律。如某一程序段运行过程中产生的计算结果需要打印输出，但此时打印机正被另一并发程序段所占用，则它只能暂停执行，并将自己挂在打印等待队列中等待打印机空闲。

（2）失去封闭性。多道程序环境下，由于多个程序共享系统资源，此时这些共享资源的状态可能被其他程序所改变，致使程序的运行受其他程序的影响而失去了封闭性。

（3）结果具有不可再现性。虽然初始条件相同，但由于程序的工作环境可能被外界因素（其他程序）所改变，故程序在不同的运行过程中所得的结果可能不同。

总之，并发执行的程序段对系统软、硬件资源的共享和竞争导致了程序的执行结果受其执行速度的影响，即程序的执行速度不同，其结果也可能不同，因而必须采取措施来制约和控制各并发程序段的执行。为了控制和协调各程序段在执行过程中对资源的共享和竞争，应该确定一个描述各程序段执行过程和共享资源的基本单位。由于程序的静态性、独立性及其执行的顺序性，无法反映操作系统所应具有的程序执行的并发性和随机性和资源共享等特征，即用程序作为描述其执行过程以及共享资源的基本单位是不合适的，故提出了进程的概念。

3.1.2 进程的定义及特征

进程（process）的概念最早是由美国麻省理工学院的J. H. Sallexer于1966年提出的，是现代操作系统中最基本、最重要的概念。由于系统并发活动的复杂性以及研究进程的角度不同，产生了多种描述进程的概念。

一般来说，进程是程序的一次执行过程，它可以和其他进程并发执行。也可以说进程是程序在一个数据集合上的一次执行过程，是系统进行资源分配和调度的独立单位。

总的来讲，进程是具有一定独立功能的程序关于一个数据集合的一次运行过程。

进程通常分成两类：用户进程和系统进程。完成操作系统功能的进程称为系统进程，而完成用户功能的进程称为用户进程，它们在运行过程中对系统内的各种资源具有不同的

访问权限。

进程的引入很好地描述了程序的执行过程和并发行为，但它们之间存在实质性的区别：

(1) 程序是静态的概念，通常所说的程序是保存在磁盘上的文件，其中包括指令序列和数据；而进程是动态的概念，它用于描述程序的动态执行过程，它因创建而产生，因执行结束而消亡。

(2) 进程具有并发特征，而程序没有。由进程的定义可知，进程是并发程序在一个数据集上的执行过程，它具有并发特征；而程序是一个静态的概念，它无法描述存在于动态过程中的并发行为。

(3) 程序是进程的一个组成部分，程序、数据集和进程控制块组成了进程实体(也称为进程映像)。数据集为进程操作处理的对象，进程控制块为用于描述进程的一个数据结构，这些概念在后续部分再详细介绍。

(4) 同一个程序可对应多个进程，但所操作的数据集不同，即同一程序段在不同的数据集上运行可以构成不同的进程。

进程与程序间存在本质区别，那么进程和作业间又有什么区别呢？由第2章知道作业是用户需要计算机完成某项任务时要求计算机所做工作的集合，它包括作业提交、收容、执行和完成4个阶段；而进程是对已提交完毕的作业的执行过程的动态描述，是资源分配的基本单位。两者是截然不同的概念，两者的具体区别为：

(1) 作业为静态的，提交给计算机后，处于外存的等待队列中。作业被调度程序选中才能调入内存，此时系统为其分配进程控制块以及其他系统资源，创建相应的进程，并插入到相应队列中等待调度、执行。而进程为动态的，只要被创建，总有相应的部分处于内存中。

(2) 一个作业至少由一个或多个进程组成，但反过来不成立。

(3) 作业的概念主要用于批处理中，而进程几乎用于所有的多道系统中。

由进程的定义以及与程序、作业的区别可以看到，进程具有如下特征：

(1) 动态性。进程的实质是进程实体的一次执行过程，所以动态特性是进程的最基本特征。进程的动态性还表现在它因“创建”而产生，因“调度”而执行，因“撤销”而消亡，它具有一定的生存周期。

(2) 并发性。指多个进程实体同时存在于内存中，并在一段时间内同时运行。引入进程的概念正是为了刻画多道程序运行的这种并发特征，由此可见并发性是进程的重要特征，也是现代操作系统的重要特征。

(3) 独立性。进程是系统分配资源和调度的独立单位，也是独立运行的基本单位。

(4) 异步性。由于进程间的相互制约，使得进程以不可预知的速度向前推进，即进程实体是按异步方式运行的。

(5) 结构性。每个进程都有一个称为进程控制块(PCB)的数据结构，该数据结构用来保存进程的相关信息。

(6) 相关性。进程在运行过程中可能会与系统中的其他进程发生直接或间接的关系。

虽然进程概念的引入很好地刻画了程序执行的动态过程和并发行为，但进程的描述、管理和调度等过程不但增加了系统的空间开销和时间开销，也增加了系统的复杂程度。

3.2　进程的描述

3.2.1　进程的组成

一个进程是一个程序在某个数据集上的一次执行过程，是分配资源的基本单位。那么进程由哪些部分组成呢？进程至少应包括被执行的程序或程序集以及要处理的数据集合，同时进程运行还要涉及存储程序和数据集的内存和保存过程调用、传递参数的堆栈以及操作系统用于监视、控制和管理程序运行的各种属性，这些属性的集合就是下面要介绍的进程控制块。

综上所述，一般一个进程由3部分组成：进程控制块(Process Control Block，PCB)、有关程序段和相应的数据结构集。进程的物理组成也称为进程映像。

PCB包括有关进程的描述信息、控制信息和资源信息等，是进程动态特征的集中反映。系统根据进程的PCB而感知进程的存在，并通过PCB控制和管理进程。PCB是进程在系统中存在的唯一标志，一般常驻内存或部分常驻内存。为了保证PCB的安全，一般PCB存储在系统数据区中。

进程的程序段描述进程所要完成的功能。而相应的数据结构集是程序执行时所涉及的工作区和操作对象，包括用户数据、用户堆栈区和系统堆栈等。在大部分多道操作系统中，这两部分内容一般放在外存中，在进程执行时再调入内存。

在一个进程的生命周期内，它所包括的PCB和相应的数据结构集中的内容是不断变化的，从而反映了进程的动态特征。

进程的组成示意图如图3.4所示，其中右图为两个进程包括同一程序段，但每个进程处理的数据集不同。反映这种情况的最好例子是可再入程序，它可被多个程序所共享，但处理的数据对象是不同的。

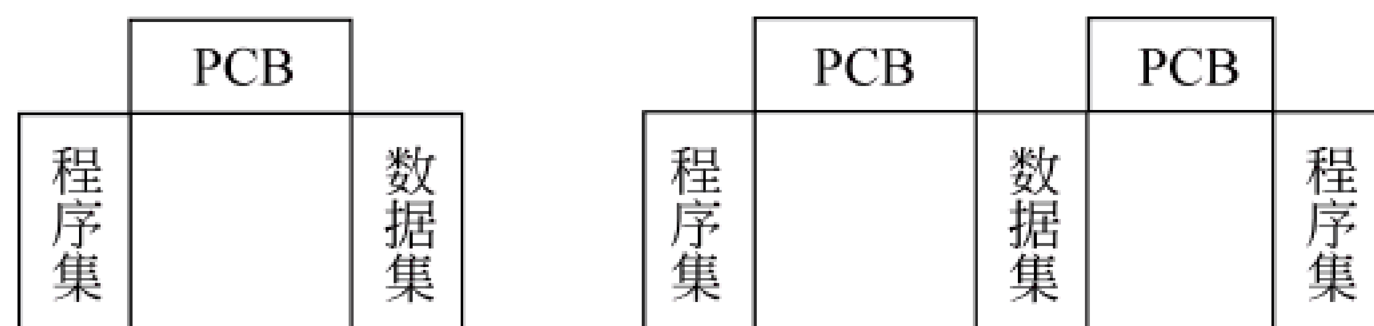


图3.4　进程的组成示意图

可再入程序也称为纯过程或纯代码，是指能够被多个进程共享的程序段，代码不因程序的执行而改变，即执行中不会改变自身代码的程序。它的主要作用就是可被多个进程共享，能被多个用户同时调用。

3.2.2　进程控制块

为了描述和控制进程的运行，系统为每个进程定义了一个数据结构——进程控制块(PCB)。它是进程的重要组成部分，它记录了操作系统所需的、用于描述进程的当前状态和控制进程运行的全部信息。操作系统就是根据进程的PCB而感知进程的存在，并依此对进程进行有效的控制和管理。PCB是进程在系统中存在的唯一标志。

系统创建一个新进程的过程就是为相应的程序建立一个PCB的过程。进程运行结束，

系统就回收其PCB,进程也随之消亡。在PCB的生存期内,它可以被操作系统中的调度程序、资源分配程序和中断处理程序等多个模块所读写和修改。显然,PCB经常被操作系统所访问,为了提高系统效率,要求PCB常驻内存;但由于内存空间有限,为了减少PCB对内存的占有量,常将PCB中常用的部分,如进程的控制信息和描述信息以及CPU现场保护信息等部分常驻内存,其他部分放在外存,运行时再装入内存。系统将所有的PCB组成若干个链表或队列,存放在操作系统专门开辟的PCB区内。

PCB主要包括如下4方面信息。

1. 进程标识信息

进程标识信息用于唯一地标识一个进程。一个进程通常有两种标识符：内部标识符和外部标识符。

内部标识符是操作系统为每个进程赋予的一个唯一的数字标识符,它通常为一个进程的序号,设置的目的主要是为了方便系统使用。

外部标识符由创建者产生,是由字母和数字组成的字符串,为用户(进程)访问该进程提供方便。

为了描述进程间的家族关系,通常还设有父进程标识和子进程标识,以表示进程间的家族关系。了解进程间的家族关系对进程管理和控制是十分重要的。因为子进程可以继承父进程所拥有的资源,例如,继承父进程打开的文件,继承父进程所分配到的缓冲区等。当子进程被撤销时,应将其从父进程那里获得的资源归还给父进程。在撤销父进程时,必须同时撤销其所有的子进程。

此外还设有用户名或用户标识号表示该进程属于哪个用户。

2. 处理机状态

处理机状态信息主要由处理机的各个寄存器内的信息所组成。进程运行时的许多信息均存放在处理机的各种寄存器中。当进程运行被中断时应保护CPU现场,即将有关寄存器中的内容保存到PCB中,以便在中断返回时能够恢复现场。这些寄存器通常包括通用寄存器(GR)、指令计数器(PC)、程序状态字寄存器和用户堆栈指针(SP)等。其中程序状态字(PSW)是相当重要的,处理机根据程序状态字寄存器中的PSW来控制程序的执行。

3. 进程调度信息

PCB中还存放一些与进程调度有关的信息。主要包括以下几项：

(1) 进程状态。标识进程的当前状态(就绪、执行和阻塞),作为进程调度的依据。

(2) 进程优先级。用于表示进程获得处理机的优先程度,优先级高的进程应优先获得处理机。

(3) 为进程调度算法提供依据的其他信息,例如,进程等待时间、进程已获得处理器的总时间和进程占用内存时间等。

(4) 事件。是指进程由某一状态转变为另一状态所等待发生的事件。

4. 进程控制信息

进程控制信息包括以下几项：

(1) 程序和数据的地址。是指组成进程的程序和数据所在内存或外存中的首地址,以便在调度该进程执行时能从其PCB中找到相应的程序和数据。

(2) 进程同步和通信机制。指实现进程同步和通信时所需采取的机制,如消息队列指

针和信号量等，它们可以全部或部分存放在 PCB 中。

(3) 资源清单。列出了进程所需的全部资源及已经分配给该进程的资源，但不包括 CPU。

(4) 链接指针。它给出了处于同一队列中的下一个进程 PCB 的首地址。

对于多道程序环境，系统中通常存在多个 PCB，这些 PCB 有的已被占用，有的处于空闲状态。为了方便对这些 PCB 的查找，通常采用链表或索引表将 PCB 管理起来。无论采用哪种管理方式，均或是将具有相同状态进程的 PCB 组成一个队列，或是根据进程进入该状态的原因而将其 PCB 组成一个队列，并设置相应的队首指针，这样可以非常方便地访问各个进程的 PCB。

链接方式把具有相同状态的 PCB 用链表连接成一个队列，这样就可以组成就绪队列、空闲队列、执行队列和若干个阻塞队列，如图 3.5 所示。其中就绪队列中的 PCB 按照相应进程优先级高低的顺序排列，而阻塞队列根据进程阻塞的原因分成多个队列，如等待 I/O 完成队列、等待分配内存队列等。空闲队列为由空闲 PCB 组成的队列。当创建一个新进程时，要从该空闲队列中取出一个 PCB；当撤销一个进程时，要将其 PCB 插入到该空闲队列中。

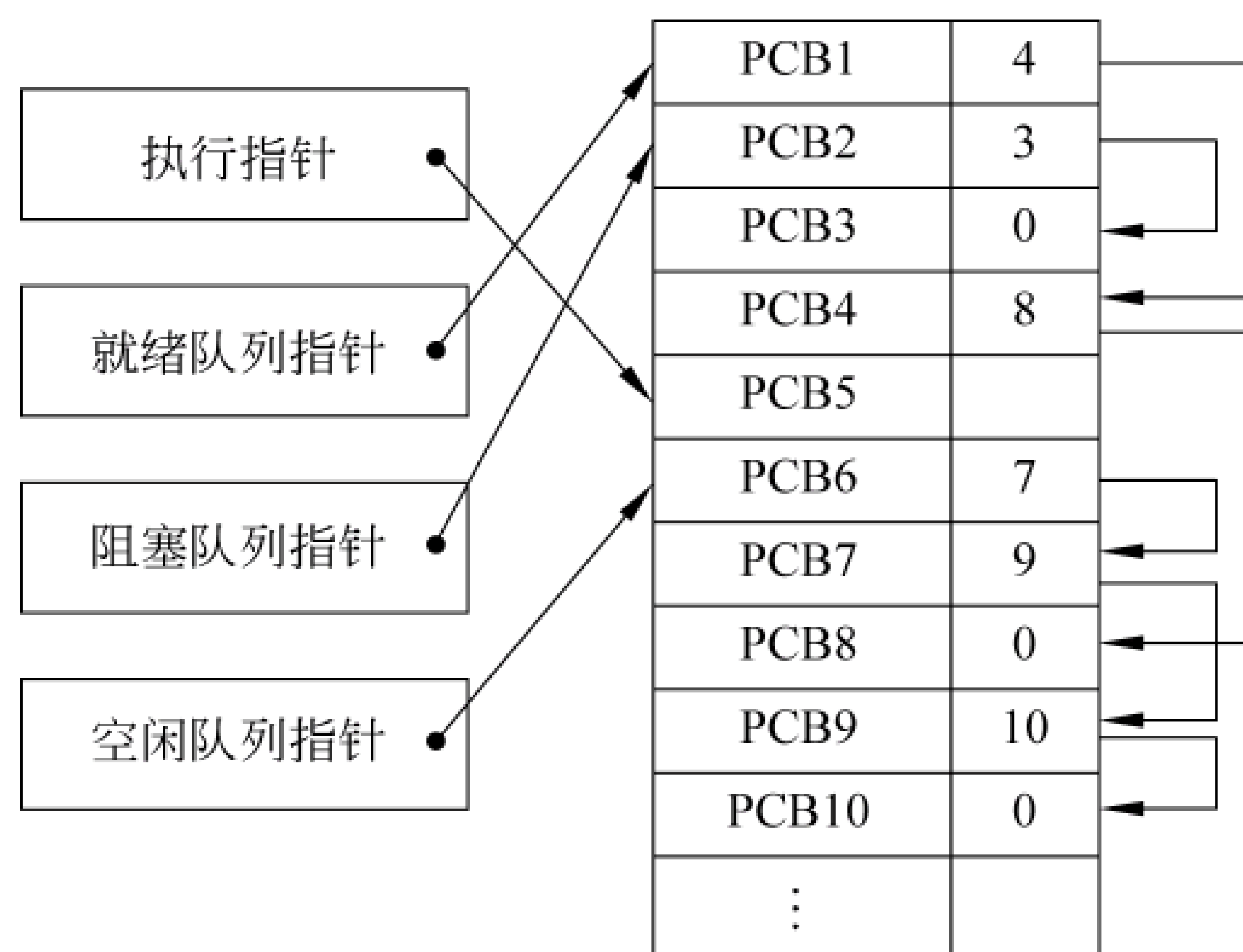


图 3.5　PCB 的链表组织方式

索引方式下，系统根据各进程的状态建立几张索引表，如就绪索引表和阻塞索引表等，并把各索引表在内存中的首地址记录在内存的一些专用单元中。在每个索引表的表目中记录了具有相应状态的某个 PCB 在 PCB 表中的地址。图 3.6 给出了索引方式的 PCB 组织。

总之，系统因 PCB 而感知进程的存在，并通过 PCB 来对进程进行调度、控制、分配资源。若进程执行结束，则通过释放 PCB 来释放进程所占用的各种资源。

3.2.3 进程上下文与进程上下文切换

1. 进程上下文

进程是在操作系统支持下执行的，进程执行时需要操作系统为其设置相应的执行环境，如系统堆栈、地址映像寄存器、程序计数器、程序状态字、打开文件表以及相关通用寄存器等，把将进程的物理实体与支持进程执行的物理环境合称为进程上下文。

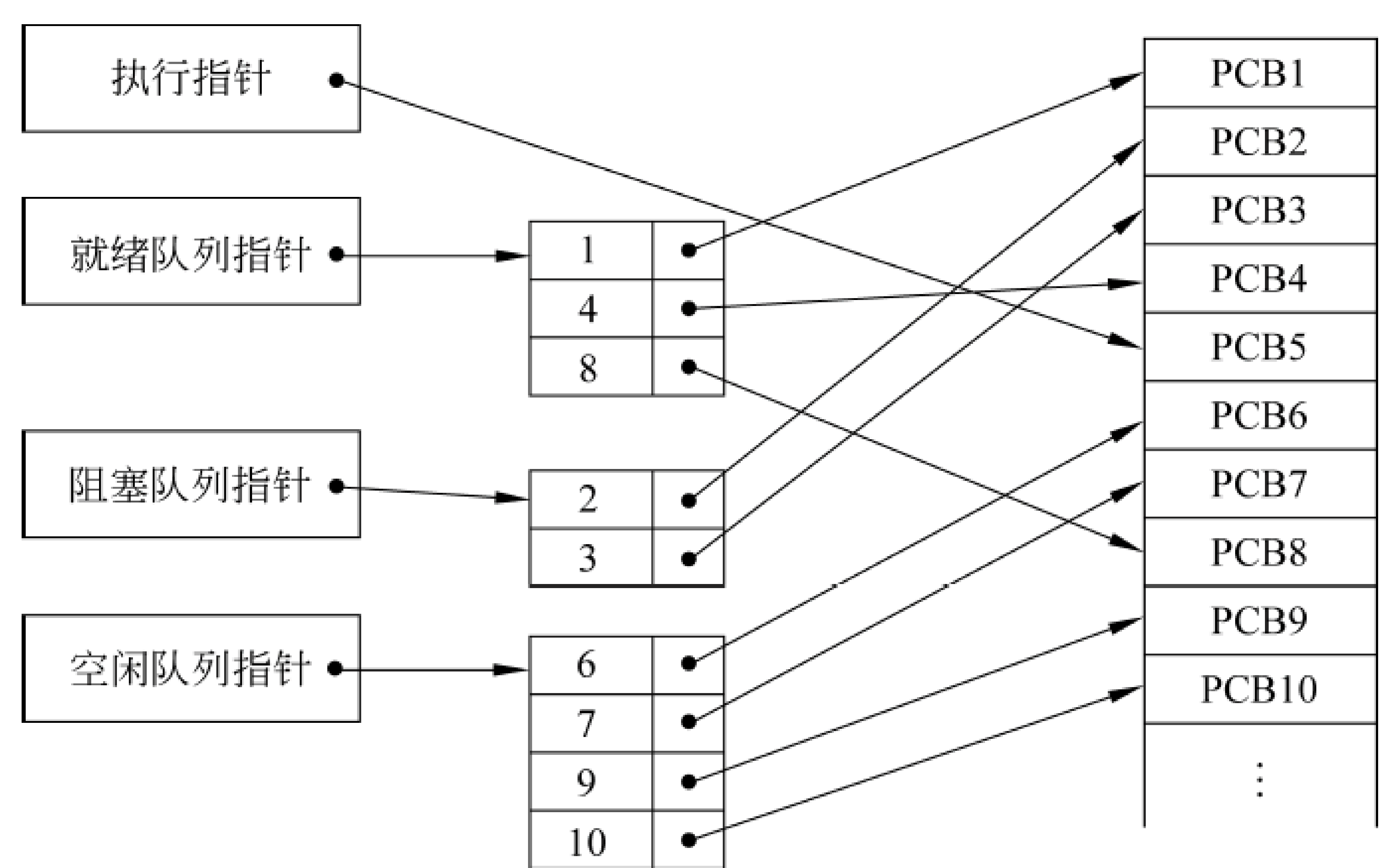


图 3.6　PCB 的索引组织方式

一般把已执行过的进程指令和数据在相关寄存器与堆栈中的内容称为上文；把正在执行的进程指令和数据在相关寄存器与堆栈中的内容称为正文；把待执行的进程指令和数据在相关寄存器与堆栈中的内容称为下文。进程上下文由上文、正文和下文构成，如图 3.7 所示。

为进一步理解进程上下文，这里给出 UNIX System Ⅴ的进程上下文的组成，如图 3.8 所示。从图中可以看出，UNIX System Ⅴ的进程上下文由用户级上下文、寄存器上下文以及系统级上下文组成。用户级上下文由进程的用户程序段部分编译而成的用户正文段、用户数据和用户栈等组成。寄存器上下文则由程序计数器(PC)、处理机状态字寄存器(PS)、栈指针和通用寄存器的值组成。其中 PC 给出 CPU 将要执行的下一条指令的虚地址；PS 给出机器与该进程相关联时的硬件状态，例如当前执行模式、能否执行特权指令等；栈指针指向下一项的当前地址；而通用寄存器则用于不同执行模式之间的参数传递等。

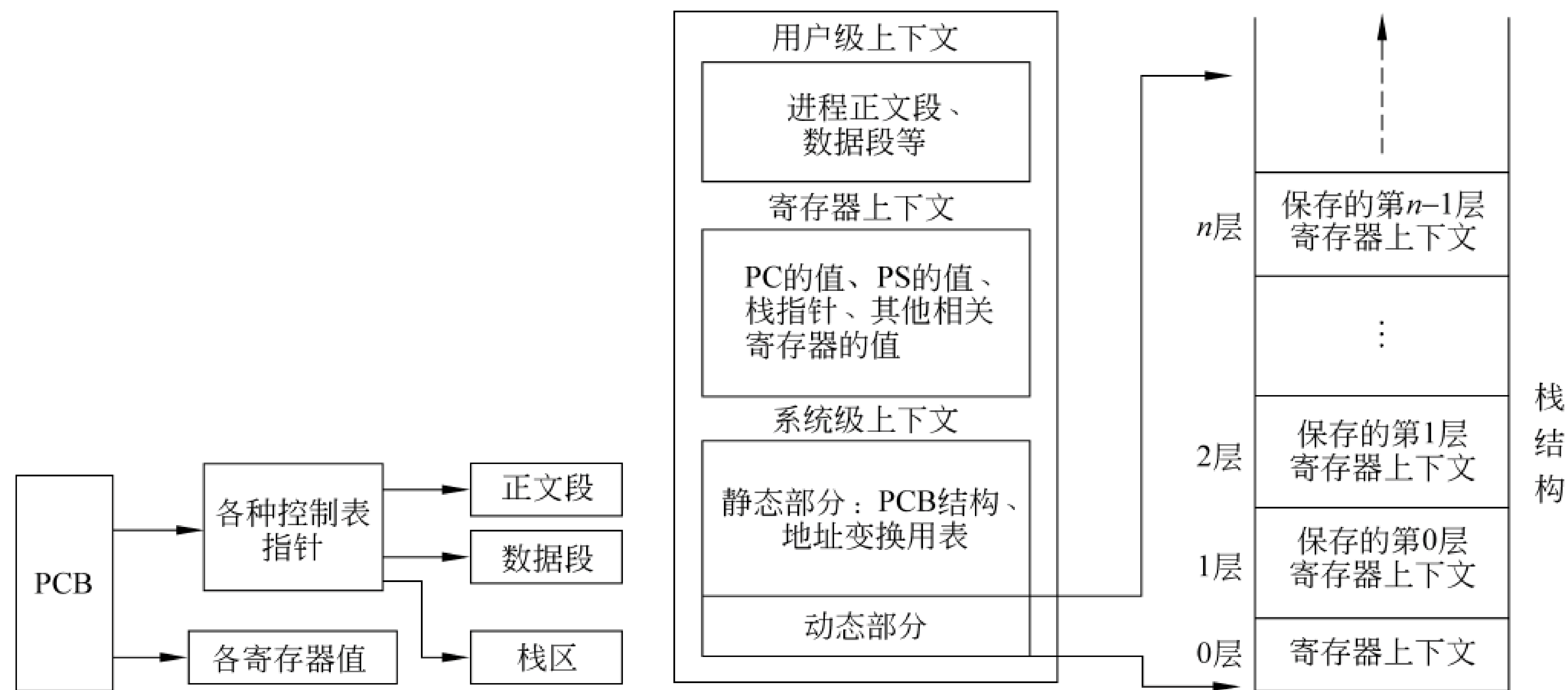


图 3.7　进程上下文的组成

图 3.8　UNIX System Ⅴ的进程上下文组成

进程的系统级上下文又分为静态部分与动态部分。这里的动态部分是指在进入和退出不同的上下文层次时，系统为各层上下文中相关联的寄存器值所保存和恢复的记录。

系统级上下文的静态部分包括PCB结构、将进程虚地址空间映射到物理空间用的有关表格和核心栈。这里，核心栈主要用来装载进程中所使用的系统调用的调用序列。系统级上下文的动态部分是与寄存器上下文相关联的。进程上下文的层次概念主要体现在动态部分中，即系统级上下文的动态部分可看成是由一些数量变化的层次组成，其变化规则符合先进后出的堆栈方式。

提出进程上下文的概念主要是为了进程上下文的切换。

2. 进程上下文切换

进程上下文切换发生在不同的进程之间而不是同一个进程内。

进程上下文切换过程分成3个步骤：

(1) 把被切换进程的相关信息保存到有关存储区，例如该进程的PCB中。

(2) 操作系统进程中有关的调度和资源分配程序执行，并选取新的进程。

(3) 将被选中进程的原来被保存的正文部分从有关存储区中取出，并送至有关寄存器与堆栈中，激活被选中进程执行。

进程上下文的切换过程如图3.9所示。

进程上下文的切换过程涉及由谁来保护和获取进程的正文的问题，也就是如何使寄存器和堆栈等中的数据流入流出PCB的存储区。另外，进程上下文切换还涉及系统调度和分配程序，这些都比较耗费CPU时间。

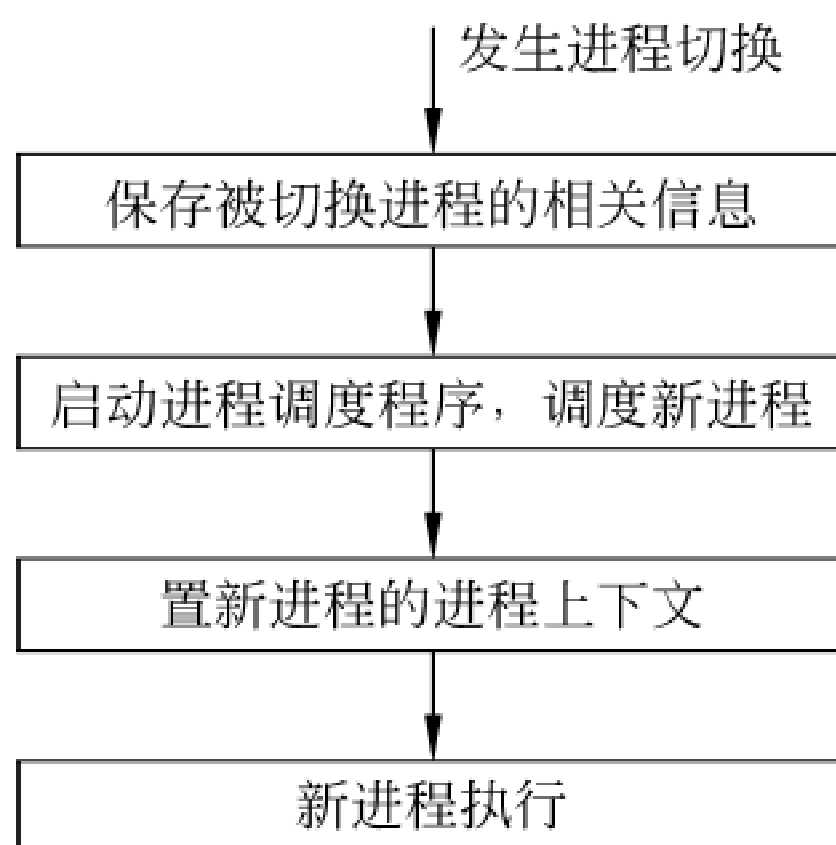


图3.9 进程上下文切换过程

3.2.4 进程空间

任一进程都有一个自己生存的地址空间，该空间称为进程空间或虚空间，该空间由一些连续或不连续的存储块所组成，主要用于存储相应的进程映像。程序的执行都在相应的进程空间内进行，组成进程的程序段、数据段以及各种控制表格等都按一定的结构和顺序排列在进程空间内。

为了确保系统安全、可靠地运行，Linux和UNIX等一些操作系统将进程空间划分成用户空间和系统空间，分别运行用户进程和操作系统核心进程(即系统进程)。为了防止用户进程访问系统空间而对系统的重要信息造成破坏，操作系统通过将程序状态寄存器等设置成不同的运行模式，即用户模式和系统模式来限制用户进程和系统进程对系统资源的访问权限，从而对系统的重要信息进行保护。

系统运行在这两种模式下的状态即为用户态(也称目态)和系统态(也称管态或核心态)。在系统态下，可以运行系统的所有指令，包括一组特权指令，可以访问所有寄存器和内存区。特权指令是指操作系统进行进程的调度和控制以及启动外围设备时所使用的一些指令。这些指令位于操作系统的内核，不允许用户程序直接使用。在用户态下，只能运行用户指令，而不能运行操作系统的特权指令，只能访问指定的寄存器和内存区。计算机系统通常将用户程序置于用户态下运行，以保证系统的安全性。

进程空间的大小只与处理机的位数有关，而与实际的内存和外存大小无关。例如，一个16位长处理机的进程空间大小为216，而32位长处理机的进程空间大小为232。

3.3 进程的状态及其转换

进程执行时的间断性决定了进程存在多种状态。一般来说，进程具有3种基本状态：就绪状态(ready)、执行状态(running)和阻塞状态(blocked)。

1. 就绪状态

进程已获得了除了CPU以外的所有资源，一旦进程获得CPU，它就可以马上进入执行状态，此时进程所处的状态就称为就绪状态。系统中可能存在多个进程处于就绪状态，通常将它们组成一个队列，称为就绪队列。

2. 执行状态

进程已获得CPU，并正在执行，即占有CPU。在单处理机系统中，某一时刻仅有一个进程占有CPU处于执行状态；在多处理机系统中，某一时刻则可能有多个进程分别占有不同的CPU而同时处于执行状态。

在某些操作系统中，进程在其执行过程中总要涉及用户程序和操作系统内核程序两部分，因此进程的执行状态又可进一步分为用户执行状态（简称用户态或目态）和系统执行状态（简称系统态、核心态或管态）。与用户程序段相对应的进程执行时说明它处于用户态，而与系统程序段相对应的进程执行时说明它处于系统态。区分用户态和系统态的主要原因是想把用户程序和系统程序分开，以利于程序的保护和共享。

3. 阻塞状态

正在执行的进程由于发生某事件而暂时无法继续执行下去，如进程申请的内存无法得到满足，申请I/O设备而I/O设备正忙，调用了某些原语等，此时进程只能放弃CPU而进入暂停状态，即进程的执行受到了阻塞，把这种暂停状态称为阻塞状态（也称等待状态）。根据进程阻塞的原因（如等待I/O设备、等待内存等），可以将处于阻塞状态的进程排成若干个队列，当相应事件发生时（如占有I/O设备的进程运行完，并释放了I/O设备），便可从相应的阻塞队列中释放一个进程使其进入就绪状态。

进程可以在上述3个基本状态之间进行转换。在分时系统中，处于执行状态的进程因用完了所分配的时间片而由执行状态变为就绪状态，并插入到就绪队列中。若处于执行状态的进程因申请某种资源（如I/O设备）而得不到满足，则该进程立即由执行状态变为阻塞状态，并插入到与该资源相对应的阻塞队列中。当某一进程执行过程中或结束时释放了某种资源（如I/O设备），则会激活位于相应的阻塞队列中的进程，使其由阻塞状态转变为就绪状态，并插入到就绪队列中。处于就绪队列中的进程被进程调度程序选中而分配CPU，则该进程将由就绪状态转变为执行状态。进程的3个基本状态之间转换如图3.10所示。

进程除了具有上述3种基本状态外，在有些系统中还引入了挂起状态，即按照一定的算法将内存中处于阻塞状态（或就绪状态）的进程暂时交换到外存（对应于挂起操作suspend），并插入到相应的挂起队列中，从而空出内存空间以调入位于外存挂起队列中将要运行的进程（对应于激活操作active），实现虚拟存储管理功能（具体见第5章）。挂起状态还可以进一步细分为外存阻塞状态（或称静止阻塞状态）和外存就绪状态（或称静止就绪状

态），把进程在内存中的阻塞状态和就绪态分别称为活动阻塞状态和活动就绪状态。图 3.11 展示了具有挂起状态进程的各种状态之间的转换。

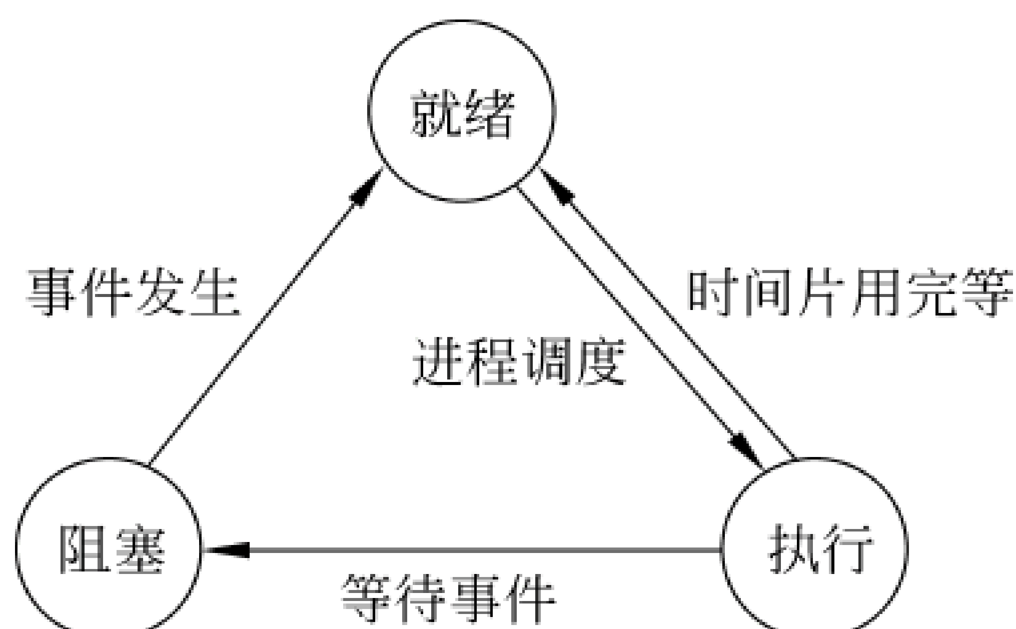


图 3.10　进程 3 个基本状态之间的转换

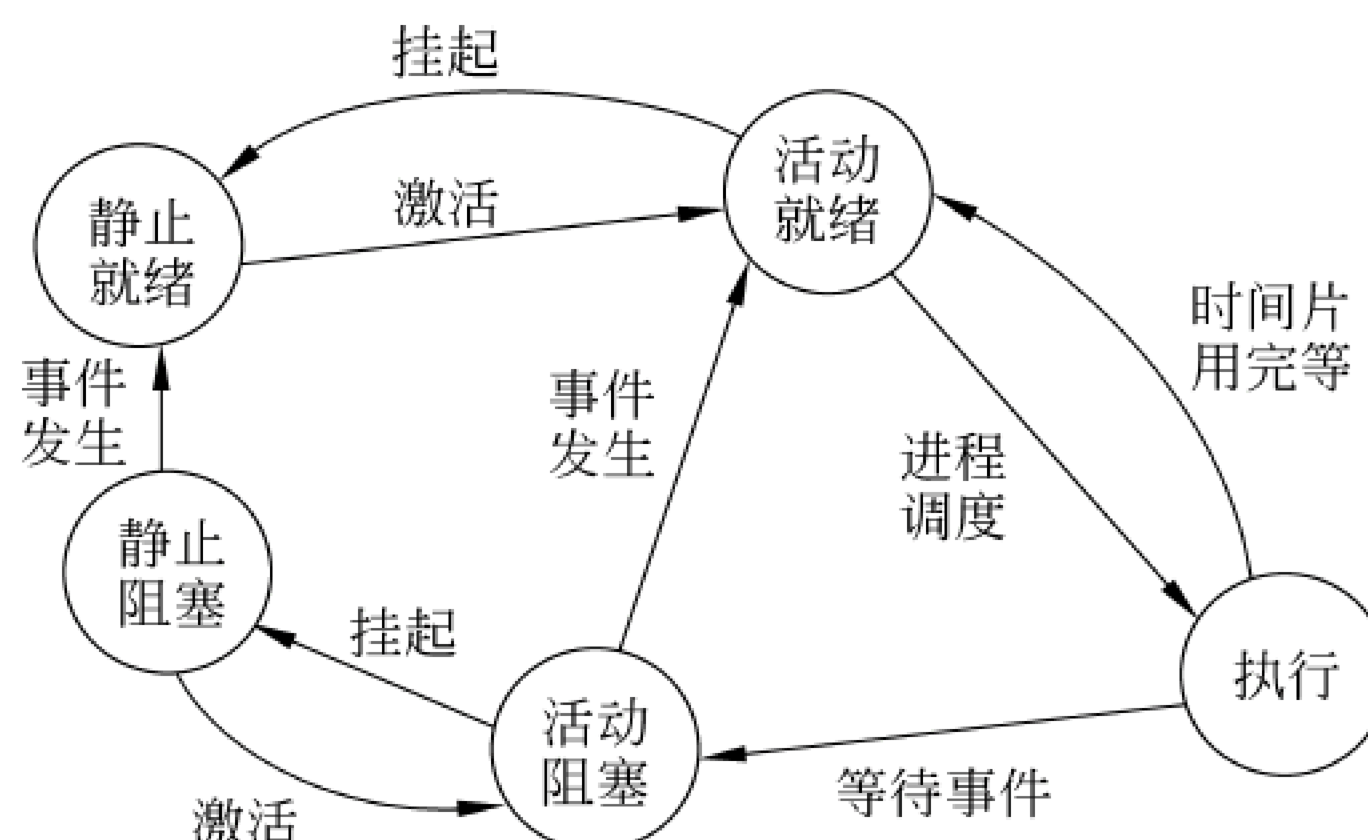


图 3.11　具有挂起状态的进程的各种状态及其转换

在目前的一些实际系统中，为了管理需要，还存在初始状态和终止状态。

当一个新进程被创建时，系统已为其分配了 PCB，填写了进程标识等信息，通常，此时的进程已拥有了自己的 PCB，但进程自身还未进入主存，即创建工作尚未完成，进程还不能被调度运行，其所处的状态就是初始状态。

引入初始状态是为了保证进程的调度必须在创建工作完成后进行，以确保对 PCB 操作的完整性。同时，初始状态的引入也增加了管理的灵活性，操作系统可以根据系统性能或主存容量的限制，推迟初始状态进程的提交。对于处于初始状态的进程，获得了其所必需的资源，以及对其 PCB 初始化工作完成后，进程状态便可由初始状态转入就绪状态。

当一个进程到达了自然结束点，或是出现了无法克服的错误，或是被操作系统所终结，或是被其他有终止权的进程所终结，它将进入终止状态。进入终止状态的进程以后不能再执行，但在操作系统中依然保留信息，这组信息供其他进程收集和使用。一旦其他进程完成了对终止状态进程的信息提取之后，操作系统将删除该进程。图 3.12 展示了增加初始状态和终止状态后进程的各种状态及其转换关系。

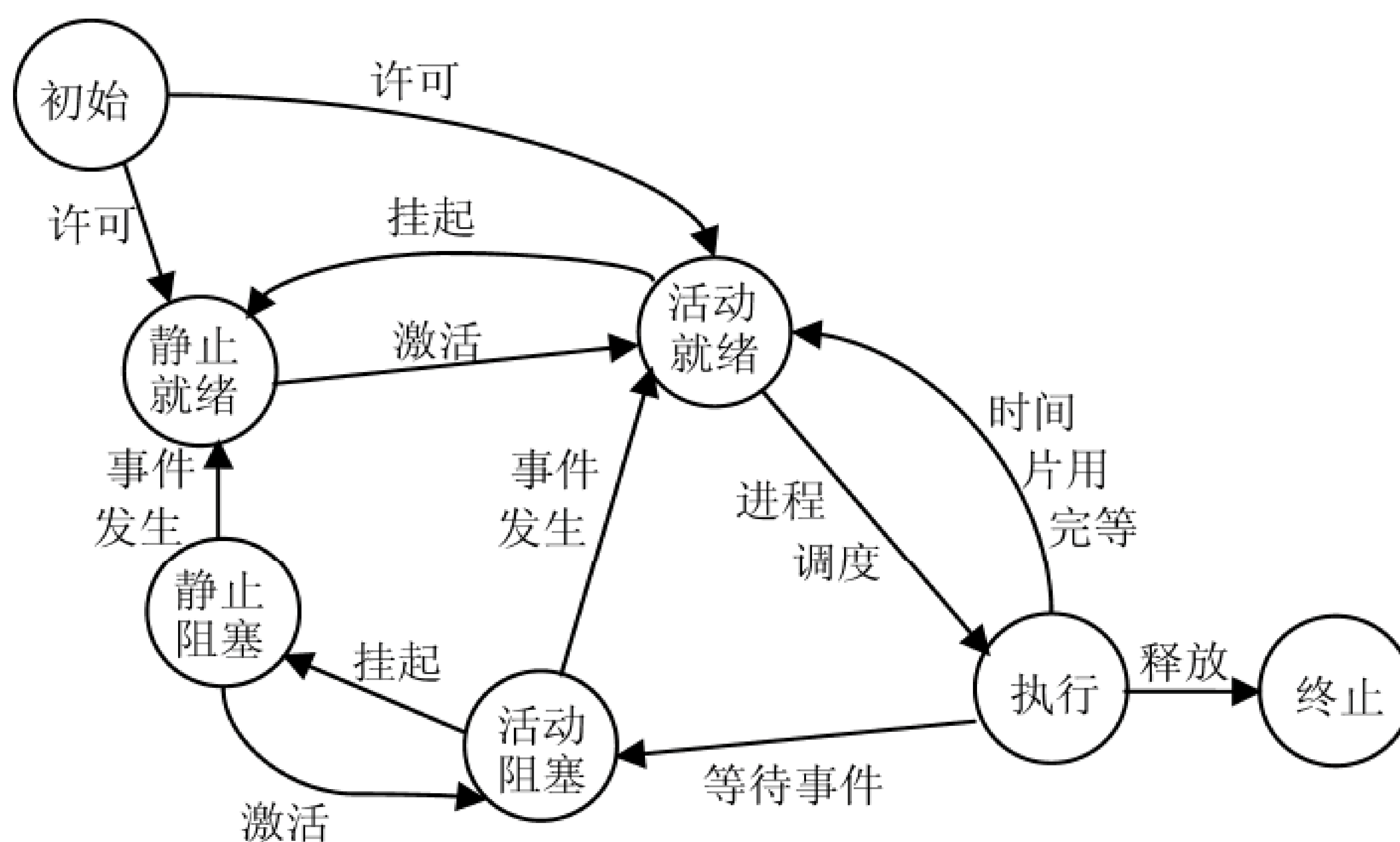


图 3.12　具有初始和终止状态的进程的各种状态及其转换

3.4 进程控制

进程控制是处理机管理中最基本的一项内容。一方面,它使用一些具有特定功能的程序段来创建、撤销进程以及完成进程各状态间的转换,从而达到多进程高效、并发、协调执行以及实现资源共享的目的。有关进程的这些操作一般在操作系统的内核中实现。另一方面,为了保护系统的关键数据不被破坏,系统根据进程的性质限制其对系统资源的访问权限,以保证系统安全、可靠地运行。

3.4.1 原语

在进程各状态之间的转换中,由就绪态转换为执行态是由进程调度程序完成的,那么其他状态之间的转换,如阻塞态转换为就绪态或执行态转换为阻塞态,又是由什么程序完成的?进程由什么程序创建、什么程序撤销?这些任务都是由称为原语的程序完成的。

原语是在系统态下执行的某些具有特定功能(如进程控制等)的程序段。原语可分成两类:一类为机器指令级的,其特点为在其执行过程中不能被中断;另一类为功能级的,其特点为作为原语的程序段不允许并发执行。

引入原语的目的是为了可靠地实现进程的通信和控制。原语均运行在系统态下,被高层软件所调用。

为了保证进程执行结果的封闭性和可再现性,通常把用做进程控制的程序段作为原语。常用的进程控制原语有创建原语、撤销原语、堵塞原语和唤醒原语等。

3.4.2 进程的创建与撤销

进程的创建和撤销分别是通过创建原语和撤销原语来实现的。

1. 进程的创建

在多道程序环境中,只有进程才能在系统中占有 CPU 而真正执行。因此,为使程序能运行,就必须为它创建进程。

引起一个进程创建另一个进程的事件主要有以下 4 类:

(1) 用户登录。在系统中,特别是在分时系统中,合法用户在终端输入登录命令后,系统将为该终端建立一个进程,并将其插入到相应的就绪队列中。

(2) 作业调度。在批处理系统中,当作业调度程序按一定的算法调度到某作业时,便将该作业装入内存,为它分配必要的资源,并立即为它创建进程,再插入就绪队列中。

(3) 提供服务。当运行中的用户程序提出某种请求后,系统将专门创建一个进程来提供用户所需要的服务,例如,用户程序要求进行文件打印,操作系统将为它创建一个打印进程,这样,不仅可使打印进程与该用户进程并发执行,而且还便于计算出为完成打印任务所花费的时间。

(4) 应用请求。根据应用进程的需要。由它自己创建一个新进程,以便使新进程以并发执行的方式完成特定任务。例如,某应用程序需要不断地从键盘终端输入数据,继而又要对输入数据进行相应的处理,然后,再将处理结果在屏幕上显示。该应用进程为使这几个操作能并发执行,以加速任务的完成,可以分别建立键盘输入进程和输出进程。

在上述4类事件中,前3类都是由系统内核根据需要创建一个新进程,而最后一类则由应用进程自己创建新进程。由此可见进程创建的方式有如下两种:

(1) 进程由系统程序模块统一创建。如批处理系统中,由作业调度程序为选中的用户作业创建PCB。这些由系统统一创建的进程间的关系是平等的。

(2) 进程由父进程创建。这种情况发生在层次结构的系统中,此时子进程隶属于父进程,并继承其父进程所拥有的资源。

操作系统一旦发现了要求创建新进程的事件后,便创建一个新进程。

创建一个新进程的过程由以下4个步骤组成。

(1) 为新进程申请一个空白PCB。为新进程申请获得唯一的数字标识符,并从空闲的PCB队列中索取一个空白PCB,将其分配给新进程。如果没有获得空白PCB,则进程创建失败。

(2) 为新进程分配必要的资源。为新进程的程序段、数据段和用户栈分配必要的内存空间,以便其装入内存中执行。

(3) 初始化进程控制块。将新进程的一些初始信息写入PCB中,这些信息主要包括:

① 标识信息。系统分配的标识符和父进程标识符。

② 处理机状态信息。主要有程序计数器和栈指针,使程序计数器指向程序的入口地址,使栈指针指向栈顶。

③ 处理机控制信息。主要有进程的状态和进程优先级,将进程的状态设置为就绪状态,优先级通常是设置为最低优先级,除非用户以显式方式提出高优先级要求。

(4) 将新进程插入就绪队列和家族树中,如果进程就绪队列能够接纳新进程,便将新进程插入就绪,同时将该进程插入其进程家族树。

创建新进程是由创建原语来完成的。进程的创建过程如图3.13所示。

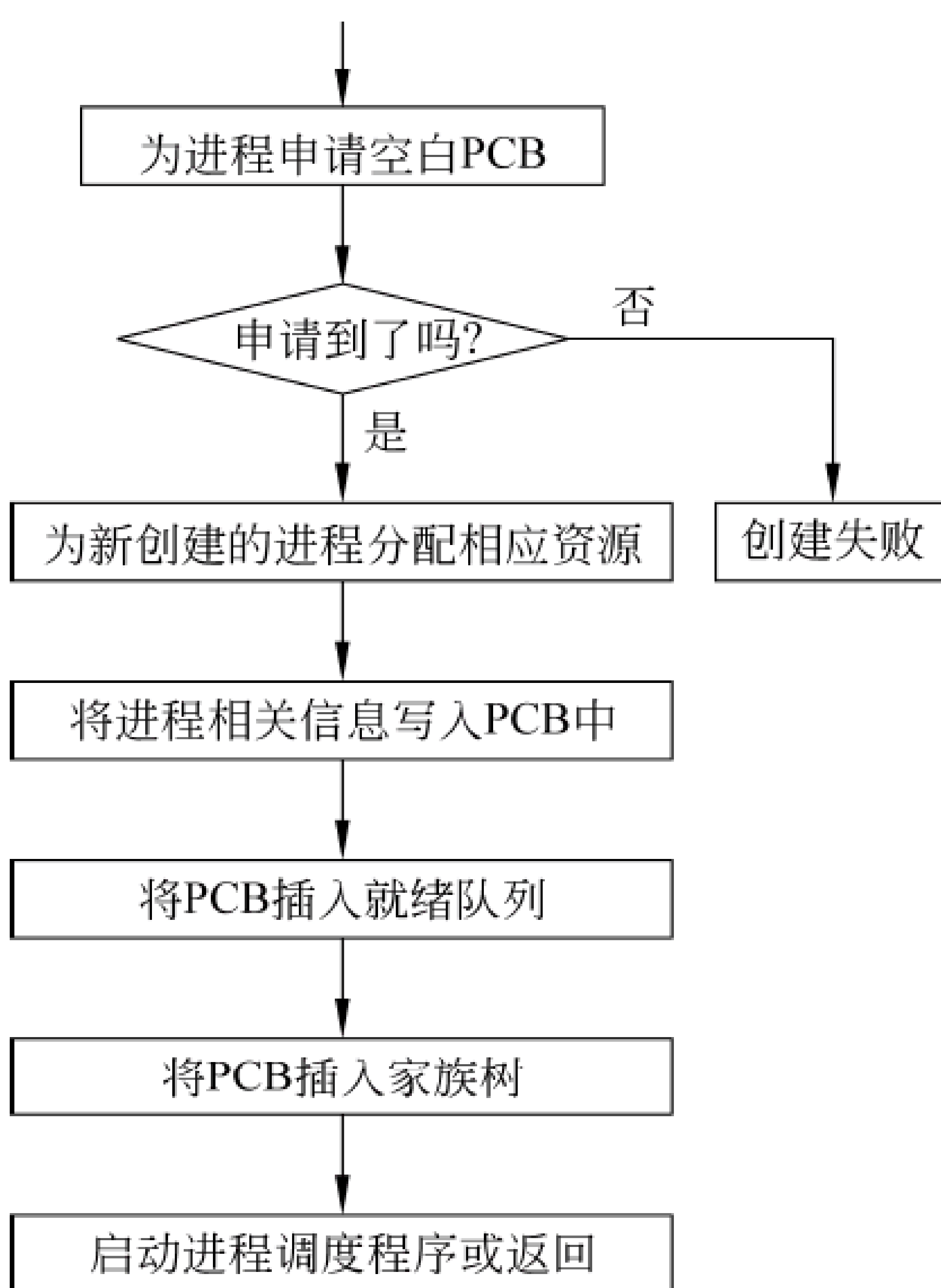


图3.13 进程的创建过程

2. 进程的撤销

在系统中,引起进程终止,从而撤销被终止的进程的事件主要有以下3类:

1) 正常结束

在任何计算机系统中,都应有一个用于表示进程已经运行完成的指示。当程序运行到指令时,将产生一个中断,通知操作系统本进程已经完成。这种情况下,进程完成了规定的任务,正常结束,撤销该进程。

2) 异常结束

在进程执行期间,由于出现某些错误和故障迫使进程停止执行,而被迫终止,从而撤销该进程。

导致这类异常事件发生的情况很多,常见的有下述几种:

(1) 越界错误。程序所访问的存储区已越出该进程的区域,从而导致出现越界错误。

(2) 保护错。进程试图去访问一个不允许访问的资源,或者以不适当的方式进行访问,如进程试图去写一个只读文件,从而导致保护错出现。

(3) 非法指令错。程序试图去执行一条不存在的指令,从而导致非法指令错出现。出现该错误的原因是程序错误地转移到数据区,把数据当成了指令。

(4) 特权指令错。用户进程试图去执行一条只允许操作系统执行的指令,从而导致出现特权指令错。

(5) 执行超时错。进程的执行时间超过了指定的最大值,从而引起进程执行超时错误的出现。

(6) 等待超时错。进程等待某事件的时间超过了规定的最大值,从而引起进程等待超时错误的出现。

(7) 算术运算错。进程试图去执行一个被禁止的运算,例如被0除,从而引起进程算术运算错误的出现。

(8) I/O故障。这是指在I/O过程中发生了错误等。

3) 外界干预

外界干预是指进程应外界的请求而终止进程执行,撤销该进程。这些干预主要有以下3种:

(1) 操作员或操作系统干预。由于某种原因,例如发生了死锁,由操作员或操作系统终止该进程。

(2) 父进程请求。由于父进程具有终止自己的任何子孙进程的权力,因而当父进程提出请求时,系统将终止相应的进程。

(3) 父进程终止。当父进程终止时,操作系统也将它的所有子孙进程终止。

撤销一个进程的工作是由撤销原语来完成的。撤销原语的工作过程如下:

(1) 检查PCB进程链或进程家族树,确定要释放的进程是否存在,若找不到被撤销进程的PCB则转入异常处理。

(2) 若找到了要撤销进程的PCB,查看该进程是否有子进程,如果有转到(1)执行。

(3) 释放该进程所占用的所有资源。

(4) 将释放的PCB插入到空闲PCB队列。

进程的撤销过程如图3.14所示。

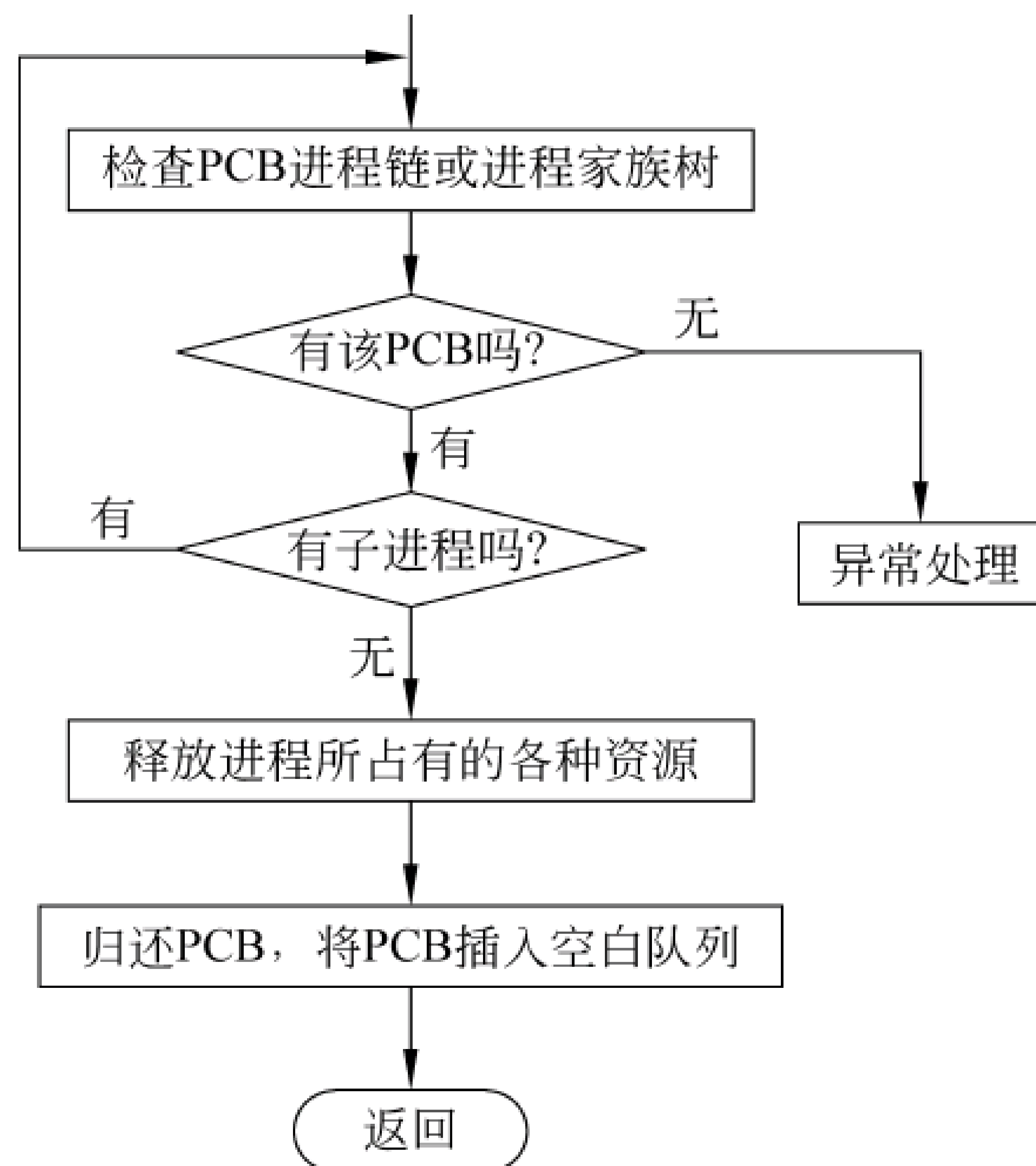


图3.14 进程的撤销过程

3.4.3 进程的阻塞与唤醒

1. 引起进程阻塞或唤醒的事件

在系统中,引起进程阻塞或唤醒的事件主要有以下几类。

1) 请求系统服务

当正在执行的进程请求操作系统提供服务,但由于某种原因,操作系统并不立即满足该

进程的要求时,该进程只能转变为阻塞状态来等待。例如,一个进程请求打印机,而系统已将打印机分配给其他进程而不能分配给请求进程,这时请求者进程只能被阻塞,仅在其他进程释放出打印机的同时,才将请求进程唤醒。

2) 启动某种操作

当进程启动某种操作后,如果该进程必须在该操作完成之后才能继续执行,则必须先使该进程阻塞,以等待该操作完成。例如,进程启动了某I/O设备进行数据输入,如果只有在数据输入这一I/O操作完成后进程才能继续执行,则该进程在启动了I/O操作后,便自动进入阻塞状态去等待;在I/O操作完成后,再由中断处理程序或中断进程将该进程唤醒。

3) 新数据尚未到达

对于相互合作的进程,如果其中一个进程需要先获得另一(合作)进程提供的数据后才能对数据进行处理,则只要其所需数据尚未到达,该进程就只有(等待)阻塞。例如,有两个进程,进程A用于输入数据,进程B对输入数据进行加工。假如A尚未将数据输入完毕,则进程B将因没有所需的处理数据而阻塞;一旦进程A把数据输入完毕,便可去唤醒进程B。

4) 无新工作可做

系统往往设置一些具有某特定功能的系统进程,每当这种进程完成任务后,便把自己阻塞起来以等待新任务到来。例如,系统中的发送进程,其主要工作是发送数据,若已有的数据已全部发送完成而又无新的发送请求,这时(发送)进程将使自己进入阻塞状态;仅当又有进程提出新的发送请求时,才将发送进程唤醒。

进程的阻塞过程和唤醒过程分别由阻塞(block)原语和唤醒(wakeup)原语完成。阻塞原语实现进程从执行状态到阻塞状态的转变。唤醒原语实现进程从阻塞状态到就绪状态的转变。

2. 进程阻塞过程

当一个处于执行状态下的进程期待某一事件(如缓冲区空、设备就绪等)发生,但事件尚未发生时,进程便调用阻塞原语阻塞自己,从而使其进入阻塞状态。此时应先中断CPU,并保存CPU现场到其PCB中,然后置该进程为"阻塞"状态,将其PCB插入到相应事件的阻塞队列中,再转到进程调度程序以选择新的就绪进程投入运行。

进程的阻塞过程如图3.15所示。

3. 进程唤醒过程

当阻塞队列中的进程所等待的事件发生时,等待该事件的进程将被唤醒。此时,唤醒原语首先将被唤醒的进程从阻塞队列中摘下,将之置为就绪状态,并插入到就绪队列中,然后唤醒原语或返回被中断的程序或转向进程调度,以便让调度程序选择一个合适的进程进入执行状态。唤醒一个进程有两种方法:

(1) 系统进程统一控制事件的发生并通过唤醒原语通知(唤醒)被阻塞的进程,使之进入就绪队列。

(2) 被阻塞的进程也可由事件发生进程唤醒。

由此可见,唤醒原语既可由系统进程调用,也可由事件发生进程调用。但要注意,进程自己不能唤醒自己。

进程的唤醒过程如图3.16所示。

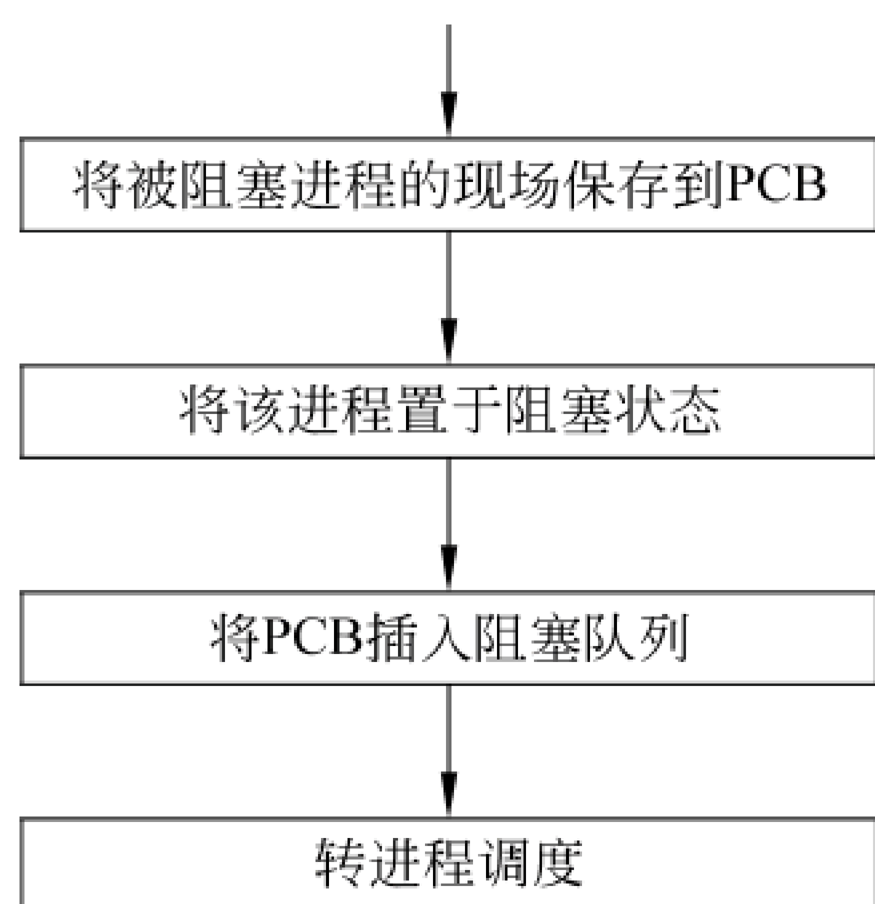


图 3.15 进程的阻塞过程

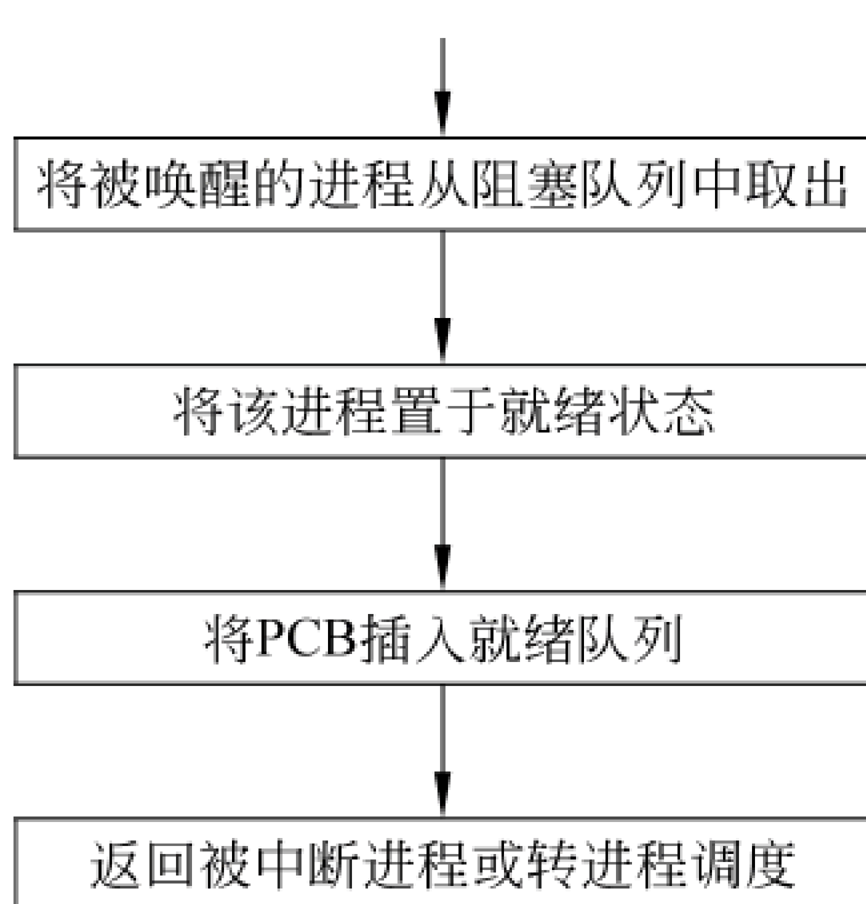


图 3.16 进程的唤醒过程

应当指出，阻塞原语和唤醒原语是一对作用刚好相反的原语。因此，如果在某进程中调用了阻塞原语，则必须在与之相合作的另一进程中或其他相关的进程中安排唤醒原语，以能唤醒阻塞进程；否则，被阻塞进程将会因不能被唤醒而长久地处于阻塞状态，从而再无机会继续运行。

3.4.4 进程的挂起与激活

在一个计算机系统中，如果引进了进程的挂起状态，那么就存在由活动就绪状态（或活动阻塞状态）转变为静止就绪状态（或静止阻塞状态）和由静止就绪状态（或静止阻塞状态）转变为活动就绪状态（或活动阻塞状态）的过程，这两个过程分别由挂起原语 suspend 和激活原语 active 来完成。

1. 进程的挂起

引起进程挂起的事件主要有以下两个：

（1）用户进程请求将自己挂起。

（2）父进程请求将自己的某个子进程挂起。

当出现了引起进程挂起的事件时，系统将利用挂起原语将指定处于活动就绪状态或活动阻塞状态的进程挂起。

挂起的执行过程如下：

（1）将被挂起进程的状态由活动阻塞状态转换为静止阻塞状态，或由活动就绪状态转换为静止就绪状态。

（2）该进程的 PCB 复制到某指定的内存区域，这样做的目的是为了方便用户或父进程考查该进程的运行情况。

（3）若被挂起的进程正在执行，则转向调度程序重新调度。

进程的挂起过程如图 3.17 所示。

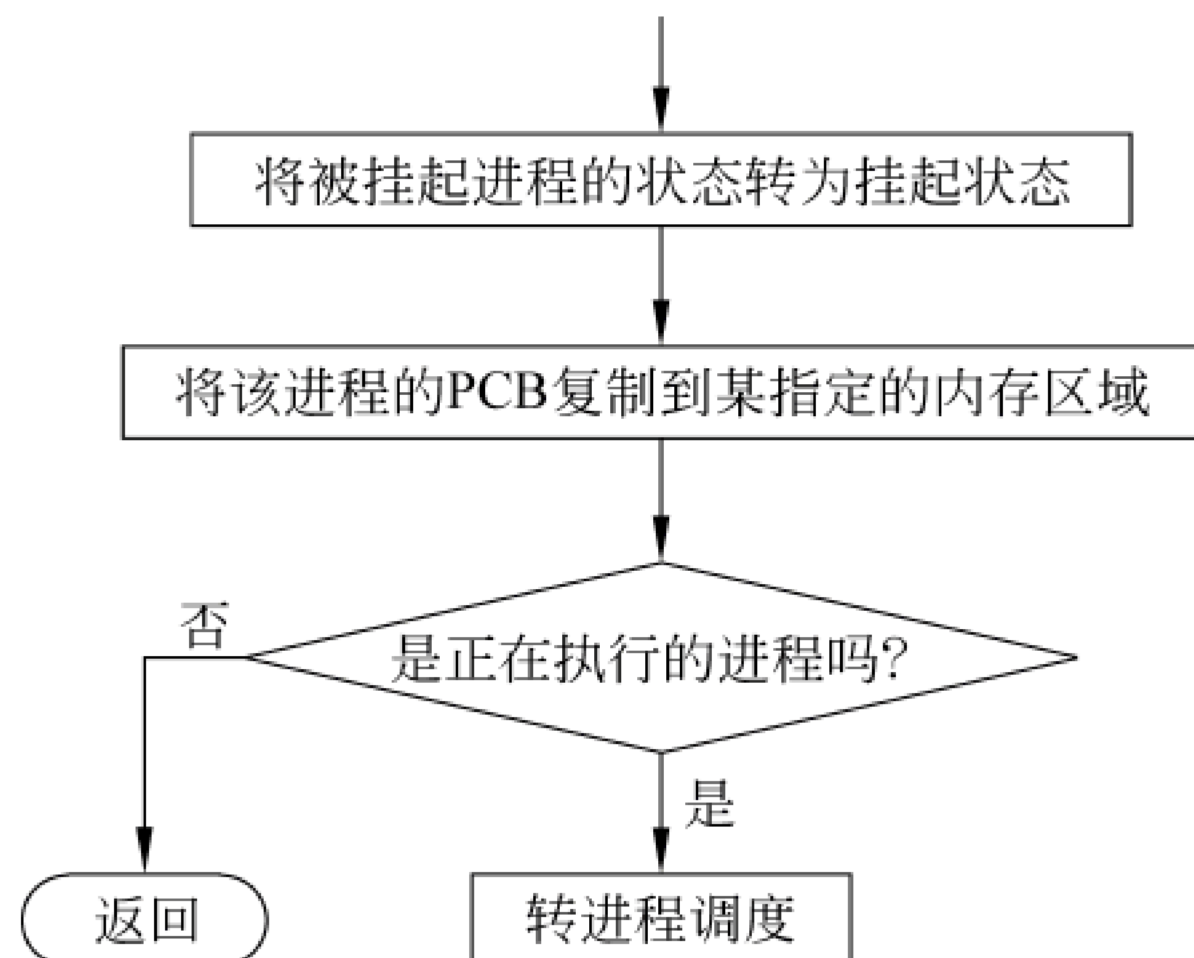


图 3.17 进程的挂起过程

2. 进程的激活过程

引起进程激活的事件主要有以下两个：

(1) 父进程或用户进程请求激活指定进程。

(2) 若该进程驻留在外存而内存中已有足够的空间时,激活该进程。

当发生激活进程的事件时,系统利用激活原语将指定进程激活,即将该进程由静止就绪状态(或静止阻塞状态)转换为活动就绪状态(或活动阻塞状态)。

激活的执行过程如下:

(1) 激活原语先将进程从外存调入内存。

(2) 将该进程插入就绪队列。

(3) 如果系统采用的是抢占时优先级调度方式,应检查是否要进行重新调度,即由调度程序将被激活进程与当前进程进行优先级的比较,如果被激活进程的优先级更低,就不必重新调度;否则,立即剥夺当前进程的运行,把处理机分配给刚被激活的进程。

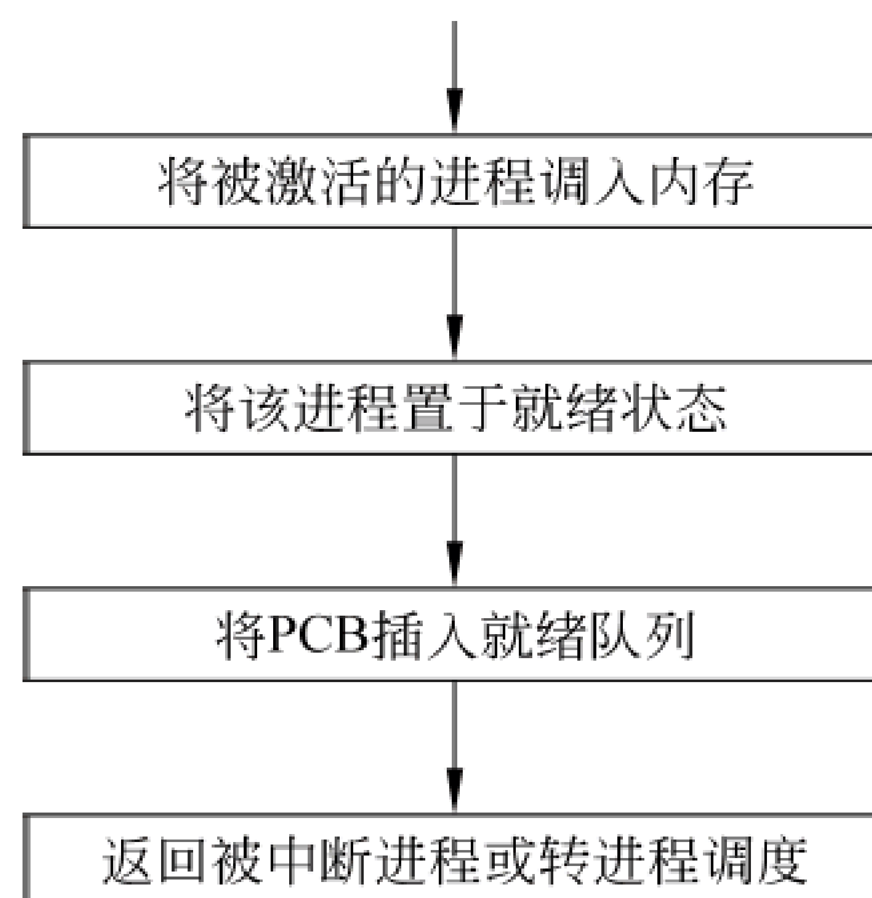


图 3.18 进程的激活过程

进程的激活过程如图 3.18 所示。

3.5 线 程

自从进程的概念提出以来,在操作系统中它一直作为拥有资源和独立运行的基本单位。到 20 世纪 80 年代中期,人们又提出了比进程更小的、能独立运行的基本单位:线程(thread),目的是为了提高系统内程序执行的并发程度,减少进程切换所需时间,从而提高系统的处理能力。特别是多处理机技术的迅速发展,线程能更好地刻画程序执行的并行特性,充分发挥多处理机的优势,所以现代操作系统中均引入了线程机制,以提高操作系统的性能。

3.5.1 线程的基本概念及分类

由以前的叙述中可知,进程为拥有资源的独立单位,也是一个可以独立调度和分派的基本单位,这是进程的两个基本特征。进程作为资源的拥有者,在进程的创建、撤销和切换过程中必然要进行资源的分配和回收、进程上下文的切换等,系统为此要付出较多的时间和空间开销。为了减少这些开销,就要求进程切换的频度不宜太高,这样系统内进程的数量就受到了限制,也就限制了程序并发执行程度的提高,这严重影响了操作系统的性能。

如何能使多个程序更好地并发执行,同时又不至于过多地增加系统开销,已成为现代操作系统设计的目标。那么能否在不切换进程上下文的情况下将程序内的并发性挖掘出来呢?这就引入了线程的概念。

线程是一个进程内的基本调度单位,也称之为轻权进程(Light Weight Process,LWP),它可以由操作系统内核控制,也可以由用户程序来控制,是程序中的一个单一的顺序控制流。

在多线程操作系统中,一个进程通常包括多个线程,即意味着一个程序内的多条语句同时执行。各个线程均可作为独立调度和分派 CPU 的基本单位,它仅拥有少量必不可少的

系统资源(如寄存器等),但它继承其所属进程的所有资源,这样线程切换时仅涉及少量寄存器和堆栈等。所以线程切换迅速,是系统内开销最小的执行实体。

下面看一下进程与线程的区别,以加深对线程概念的理解和认识。

(1) 进程为资源分配的基本单位,所有与该进程有关的资源(如打印机、缓存等)都被记录在PCB中,以表示该进程拥有这些资源或正在使用它们。同样,在现代操作系统中进程也是处理机分配的基本单位。有时,为了对进程的上述两个特性进行区分,规定线程为操作系统的基本调度单位,而进程则为系统资源的拥有者。

(2) 当进程发生调度时,它拥有一个完整的虚拟地址空间,但不同的进程拥有不同的虚拟地址空间;而同一进程内的不同线程共享其所属进程的同一地址空间。

(3) 线程只由相关堆栈、寄存器和线程控制块(Threads Control Block,TCB)组成,寄存器用来存储线程内的局部变量,但不能存储其他线程的相关变量,TCB块则用于记录线程的有关信息,起到与PCB相对应的一些功能。

(4) 进程切换时涉及有关资源指针的保存以及地址空间的变化;而同一进程内的各线程共享其所属进程的资源和地址空间,切换时无须保存资源,无地址空间变化,从而减少了操作系统的开销。

(5) 进程的调度与切换由操作系统内核完成,而线程的调度既可由操作系统完成,也可由用户完成。

(6) 在多线程操作系统中,线程是系统内的执行实体,而进程不是。

(7) 一个进程内的各个线程以及不同进程内的各个线程均可并发执行,在多处理机系统中它们可以被分派到不同的CPU上并行执行。

线程有两种基本类型:用户级线程和系统级线程(核心级线程)。有的操作系统使用用户级线程,有的操作系统使用系统级线程,如Windows NT和OS/2,有的操作系统则混合使用用户级线程和系统级线程,如Linux和Solaris。

用户级线程(user level threads)仅需应用程序自身来支持,线程的管理全部处于进程的用户空间中,不需操作系统的支持和参与。但操作系统提供了一个在用户空间内执行的线程库,该线程库提供线程的创建、调度和撤销等功能以及线程间的通信机制、线程上下文切换等。线程的调度算法和调度过程完全由用户自行选择和确定。线程调度过程中仅进行线程上下文的切换而不切换处理机,而且切换仅在用户栈、用户寄存器间进行,与操作系统内核无关,所以开销相当小。

核心级线程(kernel level threads)由操作系统内核进行管理,操作系统负责维护核心级线程的各种管理表格,负责线程在处理机上的调度和切换,并给应用程序提供相应的系统调用和应用程序接口,以使用户可以创建、执行和撤销线程。核心级线程可以被分派到同一处理机上并发地执行,也可以被分派到不同的处理机上并行地执行。操作系统内核既负责进程的调度,也负责进程内不同线程的调度。此时线程切换涉及操作系统内核,故其上下文切换要比用户级上下文切换的时间要长,系统开销相对较大,但仍小于进程切换的时间开销。

3.5.2 线程的状态及转换

线程有3种基本状态:执行、就绪和阻塞,它没有进程的挂起状态,即线程只是一个与

内存和寄存器相关的概念,它不会因交换而进入外存。线程在系统内生存的过程也就是在这3种基本状态之间转变的过程,而线程状态的转换由以下5种操作来触发:

(1) 派生(spawn):该操作可由进程产生,也可由线程产生。用户一般用系统调用(或库函数)产生自己的线程。一个新派生的线程具有相应的数据结构指针和变量,它们作为寄存器上下文存放在相应的寄存器和堆栈中。新派生的线程被放入到就绪队列中。

(2) 阻塞(block):若某个线程在执行过程中需要等待某个事件的发生,则被阻塞。阻塞时,寄存器上下文、PC和堆栈指针都会得到保存。

(3) 激活(unblock):若阻塞线程的事件发生,则该线程被激活并被放入到就绪队列中。

(4) 调度(schedule):选择一个就绪线程进入执行状态。

(5) 结束(finish):结束一个线程,释放相应的寄存器上下文和堆栈内容等。

线程的各个状态及其转换见图3.19。

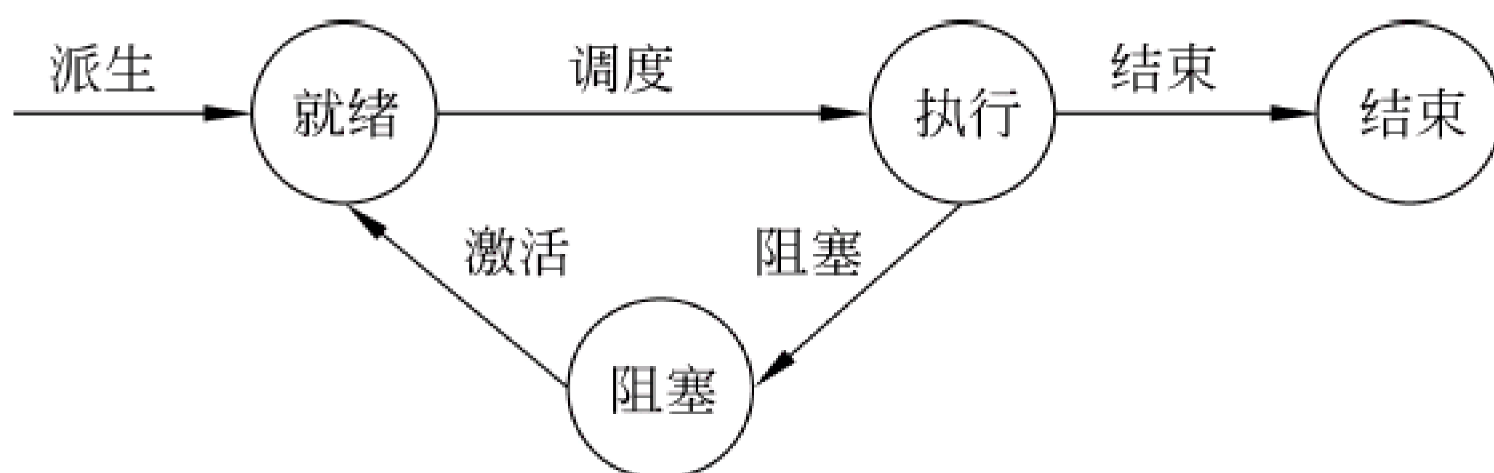


图3.19 线程的状态及其转换

3.5.3 线程的应用

虽然线程可以提高系统的执行效率,但并不适合于所有的计算机系统,如对于任务单一的实时系统、个人数字助理系统等设置线程反而会消耗更多的系统开销。

使用线程的最大优点是:当有多个任务需要处理时,可以减少处理机的切换时间,所以线程非常适合使用在多处理机系统中,这样同一用户程序可以根据不同的功能划分成不同的线程,并分派到不同的处理机上并行执行。

当用户程序可以按功能划分为多个不同的小程序段时,单处理机系统也可因使用线程而简化程序的结构和提高系统的执行效率。

下面举几个线程应用的典型实例:

(1) 服务器中的文件管理。在网络文件服务器中,对文件的访问要求由服务器进程派生出相应的线程进行处理。由于服务器可同时接受多个文件访问请求,因此系统就可以同时派生出多个线程进行处理。在多处理机环境下,各线程可被分派到不同的处理机上并行运行。

(2) 前后台处理。当系统内存在多个程序需要同时处理时,常将实时性要求不高或计算量较大的程序安排在处理机空闲的时候进行处理,即后台方式,如打印作业。

(3) 异步处理。若程序中的两部分在执行上无顺序规定,则这两部分可用线程实现。例如,在网络远程访问过程中,一个用户同时访问两个服务器;在单线程环境下,用户访问完第一个服务器并获得响应后才能访问第二个服务器;而在多线程环境下,用户可同时访问两个服务器,并同时得到响应。

3.6 Linux 的进程模型

Linux 作为一个多用户、多任务操作系统，支持多道程序设计、分时处理和对称多处理等功能。Linux 系统的进程划分为两大类，一类是系统进程，它运行在内核模式（对应于系统态、核心态或管态），执行操作系统代码，完成一些管理性的工作，例如内存分配、进程切换等；另一类为用户进程，通常在用户模式（对应于用户态或目态）下执行，并通过系统调用或在中断、异常情况发生时进入内核模式。当考虑到响应的及时性时，Linux 中的进程也可分为实时进程和普通进程。不同种类的进程在系统内享受的特权和处理方式是不一样的。

3.6.1 Linux 的进程控制块

Linux 的进程控制块（PCB）为一个由结构 task_struct 所定义的数据结构，task_struct 存放在/include/linux/sched. h 中，其中包括管理进程所需的各种信息。Linux 系统的所有 PCB 组织成结构数组形式。早期的 Linux 版本最多可同时运行进程的个数由 NR_TASKS（默认值为 512）规定，NR_TASKS 即为 PCB 结构数组的长度。近期版本中的 PCB 组成一个环形结构；系统中实际存在的进程数由其定义的全局变量 nr_tasks 来动态记录。结构数组 struct task_struct * task[NR_TASKS]={&init_task}用来记录指向各 PCB 的指针，该指针数组定义于/kernel/sched. c 中。

在创建一个新进程时，系统在内存中申请一个空的 task_struct 区，即空闲 PCB 块，并填入所需信息。同时将指向该结构的指针填入 task[]数组中。当前处于运行状态进程的 PCB 用指针数组 current_set[]来指出。这是因为 Linux 支持多处理机系统，系统内可能存在多个同时运行的进程，故 current_set 定义成指针数组。

Linux 系统的 PCB 包括很多参数，每个 PCB 约占 1KB 多的内存空间。用于表示 PCB 的结构 task_struct 简要描述如下：

```
struct task_struct{
                    ⋮
    unsigned short uid;
    int pid;
    int processor;
                    ⋮
    volatile long state;
    long priority;
    unsigned long rt_priority;
    long counter;
    unsigned long flags;
    unsigned long policy;
                    ⋮
    struct task_struct * next_task, * prev_task;
    struct task_struct * next_run, * prev_run;
    struct task_struct * p_opptr, * p_pptr, * p_cptr, * p_ysptr, * p_ptr;
                    ⋮
};
```

下面对部分数据成员进行说明。

unsigned short uid：为用户标识。

int pid：为进程标识。

int processor：标识用户正在使用的 CPU，以支持对称多处理机方式。

volatile long state：标识进程的状态，可为下列 6 种状态之一：

- 可运行状态(TASK-RUNNING)。
- 可中断阻塞状态(TASK-INTERRUPTIBLE)。
- 不可中断阻塞状态(TASK-UNINTERRUPTIBLE)。
- 僵死状态(TASK-ZOMBIE)。
- 暂停状态(TASK_STOPPED)。
- 交换状态(TASK_SWAPPING)。

long priority：表示进程的优先级。

unsigned long rt_priority：表示实时进程的优先级，对于普通进程无效。

long counter：为进程动态优先级计数器，用于进程轮转调度算法。

unsigned long policy：表示进程调度策略，其值为下列 3 种情况之一：

- SCHED_OTHER(值为 0)：对应普通进程优先级轮转法(round robin)；
- SCHED_FIFO(值为 1)：对应实时进程先来先服务算法；
- SCHED_RR(值为 2)：对应实时进程优先级轮转法。

struct task_struct *next_task, *prev_task：为进程 PCB 双向链表的前后项指针。

struct task_struct *next_run, *prev_run：为就绪队列双向链表的前后项指针。

struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_ptr：指明进程家族树中各进程间的关系，分别为指向祖先进程、父进程、子进程以及新老进程的指针。

PCB 描述中还有许多其他参数，感兴趣的读者可以参阅 Linux 系统的有关资料。

3.6.2 Linux 进程的创建和撤销

和普通的进程概念一样，Linux 中的进程也是有生命周期的，它因创建而产生，因调度而运行，因撤销而消亡。进程的创建和撤销是进程生命周期中最基本的两个操作，下面介绍这两个操作在 Linux 中的实现方式。

1. 进程的创建

Linux 启动时系统运行于核心态，此时仅创建一个 pid 号为 0 的 idle 进程；该进程会创建一个内核线程，该线程进行一系列初始化动作后最终会执行/sbin/init 文件；文件 init 运行的结果使系统的运行模式从核心态切换到了用户态，然后该线程演变为用户进程 init，其 pid 为 1。此 init 进程是一个非常重要的进程，以后系统中的一切进程都是它的后代进程。init 进程启动后系统进入空闲等待状态。

init 进程可以通过执行 fork()创建新进程。新进程的创建是通过复制老进程或当前进程来实现的。fork()函数的代码在/kernel/fork.c 中。如果 fork()执行成功，当前进程就拥有一个子进程。

创建进程的另一种方式是通过系统调用。此类系统调用有 3 个：sys_clone()、sys_vfork()和 sys_fork()。这 3 个函数最终均要通过调用 do_fork()来完成进程创建的主要工作。

2. 进程的撤销

当进程执行完毕，即正常结束时，它自己调用 exit()终止自己。当进程受到某种信号如 SIGKILL 的作用时，也是通过执行 exit()而撤销。exit()代码在/kernel/exit.c 中，其主要函数为 do_exit()。进程撤销时，一方面要回收进程所占的资源，同时也要通知其父进程。

同样，终止进程的系统调用 sys_exit()也是通过调用函数 do_exit()来实现的。do_exit()先释放进程所占的大部分资源，然后进入 TASK_ZOMBIE 状态，并调用 exit_notify()通知父进程和子进程，调用 schedule()函数重新进行进程调度。

3.6.3 Linux 进程的状态及其转换

Linux 进程共有 6 种状态，它们分别是：

(1) 可运行状态(TASK-RUNNING)。相当于进程 3 种基本状态中的执行状态和就绪状态，即表示进程正在执行或正在准备被调度执行，处于这种状态的进程才能参与进程调度。

(2) 可中断阻塞状态(TASK-INTERRUPTIBLE)。处于这种状态的进程只要阻塞的原因解除就可以被唤醒到就绪状态，并插入到就绪队列。被唤醒的原因可能是请求的资源已释放，也可能是由其他进程通过信号或定时中断来唤醒。

(3) 不可中断阻塞状态(TASK-UNINTERRUPTIBLE)。处于这种状态的进程只有资源请求得到满足才能被唤醒到就绪状态，但不能由其他进程通过信号或定时中断来唤醒。

(4) 僵死状态(TASK-ZOMBIE)。处于这种状态的子进程已经结束运行，并已释放了除 PCB 以外的部分系统资源，但要等其父进程调用 wait()函数读取该子进程的结束状态信息后才能释放所占有的其他系统资源，真正地结束运行并退出系统。

(5) 暂停状态(TASK_STOPPED)。处于这种状态的进程因被暂停执行而阻塞，通过其他进程的信号或事件才能唤醒。

(6) 交换状态(TASK_SWAPPING)。处于这种状态时，相应进程的页面可以从内存换到外存，以空出内存空间。

上述 6 种状态随着条件的变化而相互转变，各进程状态间的转变关系见图 3.20。

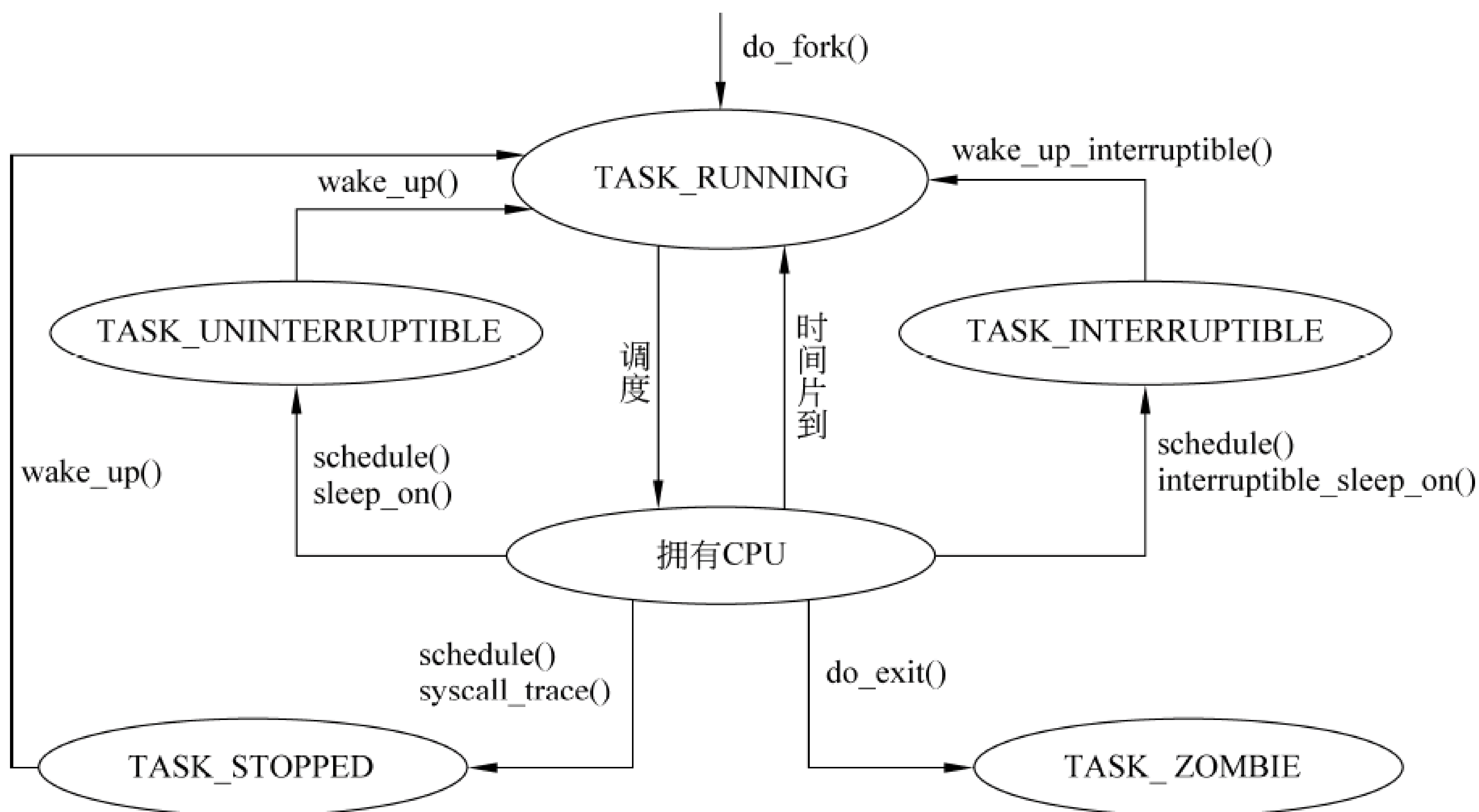


图 3.20 Linux 进程的状态及其转变

在图 3.20 中,用户进程执行 do_fork()函数完成创建进程的有关操作,并将之插入到就绪队列,在适当时候被调度程序选中,然后获得 CPU。schedule()根据系统情况完成进程调度功能。sleep_on()和 sleep_on_interruptible()使当前占有 CPU 的进程释放 CPU,并转变成阻塞状态。wake_up()和 wake_up_interruptible()实现暂停状态、阻塞状态到可运行状态的转变。do_exit()完成进程销毁的最终动作。

3.7 Linux 系统的线程机制

Linux 是一种多线程、多任务操作系统,它符合 IEEE POXIS 标准。其线程分为两种:用户线程和内核线程,在 Linux 中用在 usr/include/asm_i386/processo.h 中所定义的结构 struct thread_struct 和在 usr/include/pthread/init/pthread.h 中所定义的结构 struct pthread 分别进行描述。用户线程是在完全用户级上提供的抽象概念,是在用户程序中运行的多个控制流单元,它不需要系统内核的支持,完全在用户程序中实现。Linux 通过使用 POXIS 中的 Pthreads 线程库提供的函数就可以实现进程的创建、同步、调度和管理等操作。在用户线程中,所有用户线程存在于单一的进程中,并由该进程进行管理和调度。具体的管理调度过程对用户是透明的。显然,用户线程实现在用户级上,不使用系统调用,所以其创建、同步和撤销不需要额外的系统开销。而内核线程则不同,它不与用户线程发生联系,其管理调度由内核完成,可被系统内的其他线程访问,用户不能直接控制它。该线程的创建和撤销都要通过系统调用由内核来完成。与其他操作系统内核的实现方式不同,Linux 内核线程的实现具有更高的效率。大多数操作系统(包括 Windows NT)单独定义线程,从而增加了内核和调度程序的复杂性。而 Linux 则将线程定义为"执行上下文",是进程上下文的一部分,它仅包括少量寄存器的内容和属于线程自己的堆栈指针。线程实际上是进程的另一个"执行上下文"而已。这样 Linux 仅仅需要区分进程,其基本的调度单位仍然为进程,其调度程序仍然是进程的调度程序,在此基础上处理线程的切换。内核线程具有许多优点,例如,对于异步执行的 I/O 操作,内核可以简单地创建一个新的线程来处理这种请求,而无须提供一种特殊的机制。此外,内核线程支持并行计算和多处理器系统,线程和进程实现统一调度,I/O 效率很高。但相对于用户线程来讲,内核线程占用更多的系统资源。

3.8 本 章 小 结

进程是操作系统中最重要的概念,也是比较难理解的一个概念,它是系统资源的拥有者和处理机调度的基本单位。它用来对程序的并发特性进行动态描述,所以进程与程序以及作业存在本质上的区别。设计操作系统的目的之一就是要提高系统资源的利用率,为此引入了多道程序的概念,即允许系统内同时存在多道程序,它们共享系统资源。由于资源的有限性以及进程的并发执行,必然导致位于系统内的各个进程对共享资源的激烈竞争,操作系统通过进程调度等措施来制约、控制各进程对共享资源的合理使用以及它们的并发执行。原语是操作系统中的另一个重要概念,它是一些不允许被中断执行的程序段。为了保证系统内某些操作执行的完整性,常将完成这些操作的程序段作为原语,如进程状态间的转换就是通过一些原语来触发的,这些原语有阻塞原语、唤醒原语等。进程在系统中存在多种状

态，如执行、就绪、阻塞和挂起等。对于单处理机系统，除执行状态外，其他进程状态均以单个或多个队列的形式存在于系统中，各进程在各种队列间迁移的原因可能是系统执行了原语操作，也可能是某些事件的发生。

线程是操作系统中的一个非常重要的概念，它用来描述进程内部的并发或并行特性，是进程内的一个单一控制流，是在更细粒度上对并行特性进行描述。通常一个进程内可以包括多个线程，它们共享该进程的所有资源。这些线程可以并发执行，在多处理机系统中它们可以被分派到不同的处理机上并行地执行。由于同一进程的所有线程均享有同一用户地址空间，在处理机切换时所花的时间开销和空间开销非常少，故其切换速度比进程要快得多。

Linux 操作系统支持多任务、多线程机制，它采用一个结构数组来对进程进行描述。它的进程在系统内有 6 种状态，所以它的进程模型描述起来较为复杂。Linux 系统支持对称多处理机系统，系统内的多个线程可以被分派到各个处理机上并行地执行，从而可以显著提高系统的处理能力。

习　　题

1. 说明程序顺序执行的基本特征。
2. 说明程序并发执行的基本特征。
3. 为什么程序并发执行时会产生间断性特征？
4. 程序并发执行时为什么会失去封闭性和可再现性？
5. 解释进程的概念以及与程序、作业的区别。
6. 分别论述引入进程概念的优点和缺点。
7. 说明并行和并发的概念及区别。
8. 说明进程的主要特征。
9. 说明进程的组成部分及其具体内容。
10. 说明 PCB 的具体结构和作用以及 PCB 中所包括信息的种类。
11. 说明 PCB 的组织方式。
12. 两个不同进程的执行代码一定不相同吗？请举例说明。
13. 进程有几种基本状态？进程在各个状态之间转换的原因是什么？
14. 系统在某一时刻是否一定有一个用户进程处于运行状态？
15. 当一个进程执行出错后，它是否转变为阻塞状态？
16. 什么是原语？它有什么特点？
17. 说明进程创建和撤销时所要完成的主要工作。
18. 进程能自己唤醒自己和撤销自己吗？
19. 说明线程的概念以及引入线程概念的意义。
20. 说明线程和进程的主要区别。
21. 说明线程的种类及其特点。
22. 有了线程的概念后，那么线程和进程分别是什么的基本单位？
23. 简单举例说明线程的应用。
24. 理解 Linux 进程的创建和撤销过程。

25. 说明 Linux 系统中进程控制块的组织方式。

26. 说明 Linux 系统中线程的种类及其创建方法。

27. 已知一个求值表达式(A2+3B)/(B+5A),若 A、B 已赋值,试画出该公式求值过程的前趋图。

28. 某系统的进程状态如图 3.21 所示。a 是__①__状态,b 是__②__状态,c 是__③__状态。1 表示__④__,2 表示__④__,3 表示发生了等待事件,4 表示等待事件结束。下列情况中,当发生前者的状态转换时,__⑥__会导致发生后者的状态转换。

①②③:A. 挂起　B. 运行　C. 阻塞　D. 就绪

④⑤:A. 落选　B. 选中　C. 阻塞

⑥:A. 2－>1　B. 4－>2

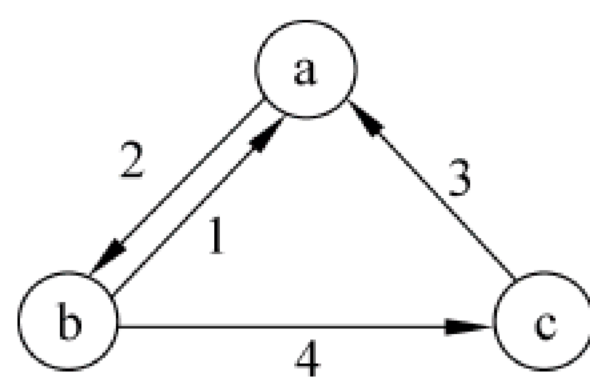


图 3.21　进程状态图

第 4 章　处理机管理

处理机(Central Processing Unit,CPU)是计算机系统的运算和控制中心,其处理能力是评价计算机系统性能的重要指标。操作系统设计的目的是增强系统的处理能力和提高系统资源的利用率,而处理机作为系统的重要资源,如何有效地进行管理和调度就成为设计操作系统时应优先考虑的问题。通过操作系统的发展史可以看出,处理机利用率的不断提高及其处理能力的不断增强是操作系统发展的重要标志。

处理机调度是多道程序系统的基础。在多道程序环境下,系统内的作业或进程的数量通常多于处理机的个数。处理机调度就是从众多的作业或进程中依据一定的准则选择若干个作业或进程,并将处理机分配给它们,其功能具体由作业(进程)调度程序来实现。该调度程序在运行时按照一定的算法准则从就绪队列中选择一个或多个作业(进程),然后将处理机分配给它(们),并尽量保证每一时刻均有一个或多个作业(进程)在使用处理机,从而可以确保计算机系统的高效运行。

4.1　分级调度

处理机是计算机系统中最重要的硬件资源,其利用率的高低直接关系到系统的使用效率。而处理机利用率的高低主要取决于处理机的有效管理,其中处理机调度算法的选择是非常关键的,而且对于不同种类的操作系统,处理机的调度算法也不尽相同。操作系统根据调度对象的不同或调度性质的差异,一般采用分级调度的方法实现处理机的有效分配,即处理机调度分 4 级:作业调度、交换调度、进程调度和线程调度。其中作业调度和进程调度存在于多道批处理系统中,交换调度、进程调度和线程调度存在于分时系统和实时系统中,而分时系统和实时系统中不存在作业调度。在一个作业的生命期中 4 级调度间的关系如图 4.1 所示。

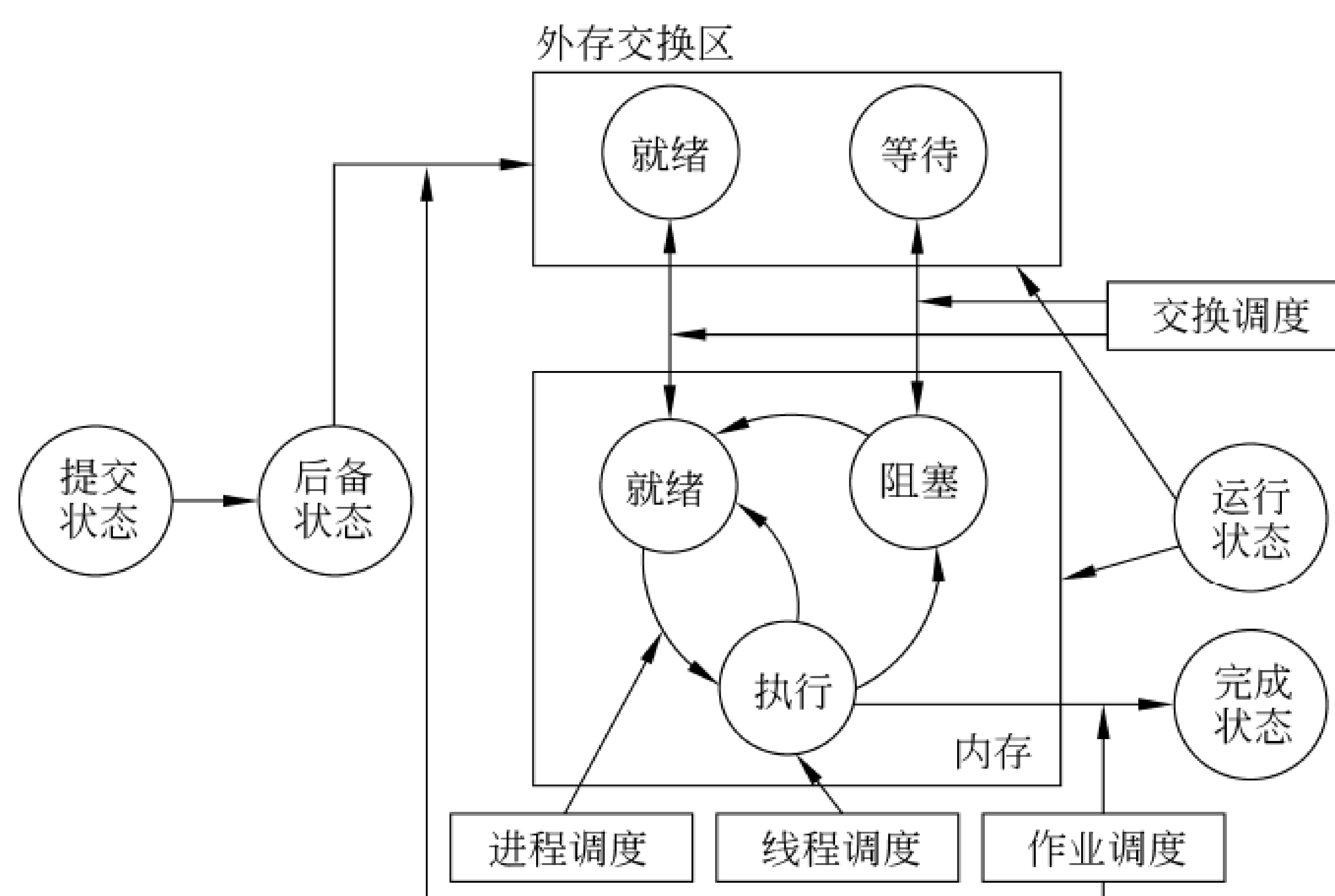


图 4.1　4 级调度关系

4.1.1　作业调度

作业调度又称高级调度或宏观调度，它是按照某种算法从后备作业队列中选择一个或多个作业装入内存，并在作业运行结束后做善后处理。完成作业调度功能的程序称为作业调度程序，它决定哪个用户作业可以进入系统进行处理，它也决定多道程序设计的程度(此外，多道程序设计的程度也受内存等系统资源规模的限制)。一旦一个作业被作业调度程序选中，给该作业分配相应的资源，系统为之创建进程并插入到相应的就绪队列中等待进程调度。当该作业执行完毕时，作业调度程序还负责收回分配给该作业的资源，做善后处理工作。

4.1.2　交换调度

交换调度也称为中级调度，其功能是在内存和外存间进行信息交换，即将内存中暂时不具备运行条件的进程挂起，将其交换到外存交换区中处于外存等待状态，空出内存空间以容纳外存中即将换入内存的、将要运行的进程，从而实现虚拟存储管理。交换调度的具体技术将在第 5 章中进行详细介绍。

4.1.3　进程调度

进程调度又称为低级调度或微观调度，和作业调度一样，进程调度也是处理机调度的重要组成部分，它协调和控制各进程对处理机的使用。相应的进程调度程序称为分派程序或低级调度程序。对于多道程序环境，系统内同时存在多个正在执行的程序，系统内也就存在多个与之相对应的进程。其中的一些进程可能已获得了除处理机以外的其他所有资源，这样就绪队列中就存在多个进程等待分配处理机。进程调度的任务就是按照一定的准则合理地将处理机动态地分配给处于就绪队列中的某个进程，使之投入运行。根据各种实际情况的不同，调度算法选择进程的准则有多种，后续部分将详细介绍。

4.1.4　线程调度

第 3 章引入了线程的概念，并介绍了线程模型。线程通常分为用户级线程和核心级线程。用户级线程由线程库进行管理，内核并不感知它们的存在，但为了能够占有处理器，用户级线程最终还是要映射成核心级线程，尽管这种映射可能是间接的，也可能使用了轻权进程。用户级线程和核心级线程调度的方式是不同的。用户级线程由线程库进行管理和调度，并运行在一个可以得到的轻权进程上，这种模式称为进程局部调度；而核心级线程则由系统内核来调度，称为系统全局调度。

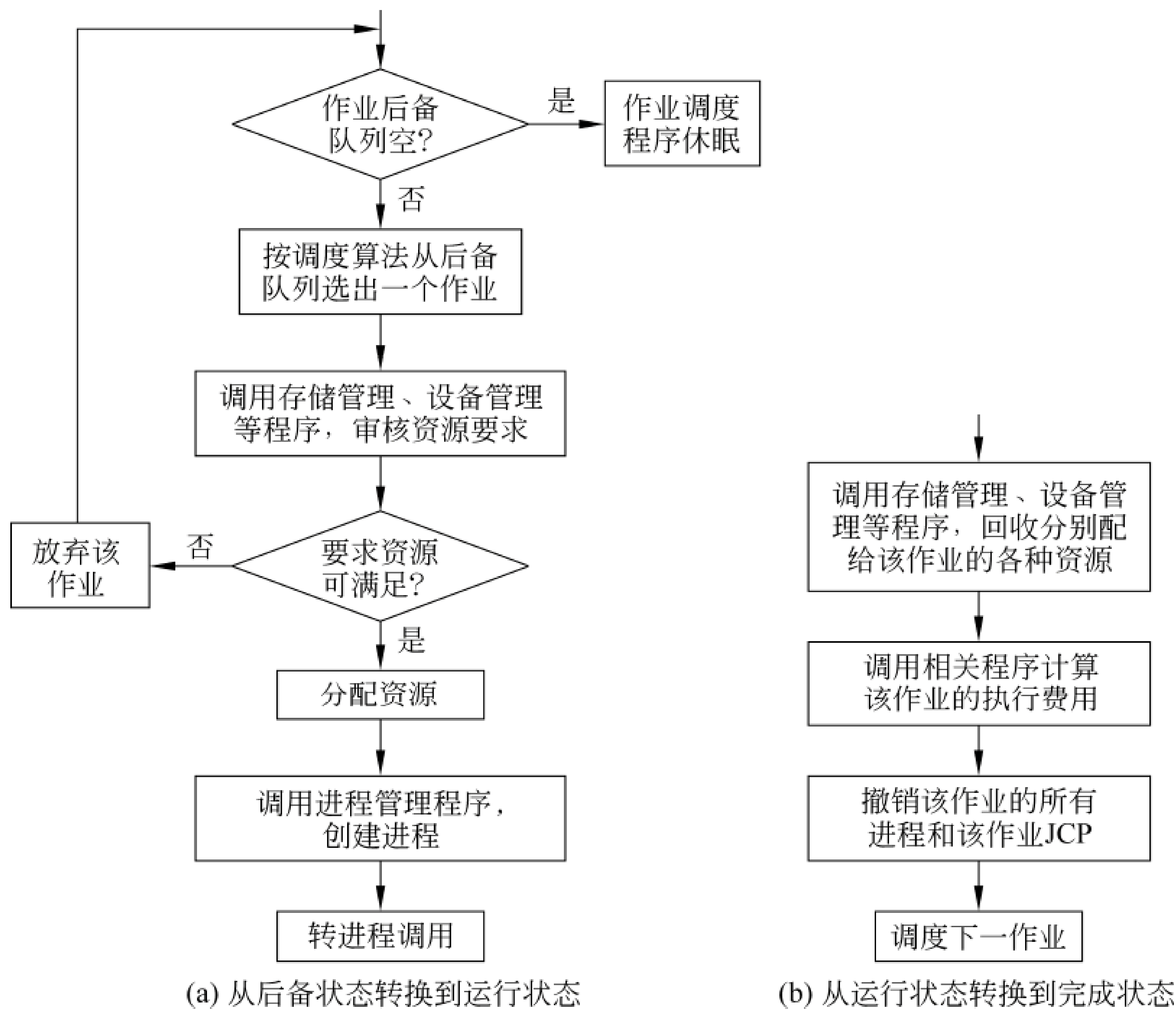
目前许多程序设计语言均支持多线程程序设计，Java 是比较典型的一个。众所周知，Java 具有很好的跨平台特性，主要是因为它有 Java 虚拟机(Java Virtual Machine，JVM)的支持。JVM 采用基于优先级的抢占式线程调度算法。所有的 Java 线程均被赋予一个优先级，JVM 将处理机分配给具有最高优先级的线程；当存在两个线程同时具有最高优先级时，采用先来先服务算法；其中优先级调度和先来先服务算法等在 4.2 节详细介绍。

4.2 作业调度和进程调度

4.2.1 作业调度

1. 作业调度的功能

作业调度主要是完成作业从后备状态转换到运行状态和从运行状态转换到完成状态。作业调度中状态的转换过程如图 4.2 所示。



(a) 从后备状态转换到运行状态　(b) 从运行状态转换到完成状态

图 4.2 作业调度中状态的转换过程

具体来说,作业调度完成的主要功能如下:

(1) 记录系统中各作业的状况,包括后备状态、运行状态和完成状态的作业状况,不包括提交状态的作业情况,因为提交状态的作业还没有完全进入系统,不存在作业控制块。作业控制块是作业由提交状态转换为后备状态时系统为作业创建的,有了作业控制块,系统就能感知作业的存在;当作业执行完毕进入完成状态之后,系统撤销作业的作业控制块并收回分配给该作业的各种资源,撤销该作业。每个作业的各阶段所需要的和分配的各种资源以及作业的状态都记录在作业控制块中,作业调度程序根据作业控制块中的相关信息对作业进行调度和管理。

(2) 选择作业。按照某种作业调度算法从后备作业队列中选择一个作业或几个作业投入运行。一般来说,处于后备状态的作业很多,但处于运行状态的作业只能有几个(由系统拥有的资源数量和作业运行时需要的资源等决定)。作业的选择主要依据作业调度算法以

及作业控制块中的信息。

(3) 分配资源。调用存储管理和设备管理等模块所提供的功能,为选中的作业分配内存、输入输出设备等资源。

(4) 为作业创建进程。为选中的作业创建进程,申请一个PCB,填入必要的控制和状态等信息,并将该进程设置为就绪状态,然后将其PCB插入到就绪进程队列中等待进程调度。

(5) 作业后续处理。完成作业正常结束或因出错而终止时的后续处理工作,例如:资源回收,输出必要信息,撤销和回收该作业的各进程PCB和JCB。

2. 作业调度的目标

上面介绍了作业调度的功能,在这些功能中,最主要的是从后备队列选择一个或几个作业来执行;在进行选择时,根据不同的目标,将会有不同的调度算法。作业调度目标如下:

(1) 对所有作业应该是公平合理的。

(2) 应使设备有高的利用率。

(3) 每天执行尽可能多的作业。

(4) 有快的响应时间。

上述目标不可能同时达到,因为它们之间有些是相互冲突的。例如,要想执行尽可能多的作业,就要优先选择执行时间短的作业进行调度,但这样就可能使得执行时间长的作业需要等待过长时间或者得不到调度,这样对执行时间长的作业来说响应时间也就增长,对它们就不公平了。因此在选择调度算法时,要结合用户的要求,综合考虑作业调度目标。

4.2.2 进程调度

在系统中存在用户进程和系统进程,这些进程数量一般是多于处理机的数量的,这样就会有进程争夺处理机的情况发生,选择哪个进程占有处理机,何时占有,以何种方式占有,就是进程调度要解决的问题。

1. 进程调度的功能

总体上来说,进程调度应完成以下功能:

(1) 记录系统中所有进程的有关信息。为了对系统内的进程进行有效的管理,系统必须掌握有关进程的所有信息(如进程当前状态、优先级、资源使用情况和被中断时的CPU现场数据等)。这些信息均记录在各个进程的PCB中,并采用链表等数据结构对处于各种状态下的进程的PCB进行组织,从而使系统能够方便、有效地管理系统内的各个进程。

(2) 确定处理机的分配原则。即按照一定的准则从就绪队列中选择一个进程,为之分配处理机,同时还要确定将处理机分配给每个进程使用的时间片的长短。分配处理机时所依据的准则即为进程调度算法,进程调度程序依此算法从就绪队列中选择一个进程,为之分配处理机,使之运行。

(3) 分配处理机。把处理机的控制权交给被选中的进程,让它开始执行,即该进程从就绪队列中摘下,并由就绪状态置为执行状态。

(4) 切换进程上下文。进程上下文即进程运行的工作环境,它包括与进程运行有关的变量和数据结构的值、各种硬件寄存器(PC、PSW和堆栈指针等)的值、进程的PCB以及程序段等。当正在运行的进程由于某种原因要退出处理机时,系统要保留被切换进程的工作上下文,以便以后切换回该进程时能顺利地恢复和执行。当某个进程被调度程序选中获得

处理机的使用权后，系统就为之装配进程上下文，并将处理机的控制权转交给该进程。

(5) 回收处理机。正在运行进程的时间片用完或因申请的资源无法得到、或在可剥夺方式下具有更高优先级的进程需要处理机，此时进程调度程序需收回处理机，然后再根据进程调度算法在就绪队列中选择一个满足条件的进程，为之分配处理机并使之投入运行。

2. 进程调度的方式

进程调度有两种方式：

(1) 非抢占式(也称为不可剥夺方式)，它的含义是：即使就绪队列中的某个进程的优先级高于正在执行进程的优先级，也要等执行进程主动让出处理机后，高优先级进程才能得到处理机。

(2) 抢占式(也称为可剥夺方式)，它是指当就绪队列中某一进程的优先级高于正在执行进程的优先级时，系统可剥夺正在执行进程的处理机的使用权(即使分配给它的时间片还没有用完等)，而使高优先级的进程抢占处理机，使之执行。

这里所讲的优先级用来表示进程占有处理机的优先程度，后面要介绍一种基于优先级的进程调度算法，就是以进程的优先级作为调度进程占有处理机的主要依据。

3. 进程调度的时机

何时进行进程调度，不仅与操作系统的性质有着密切关系，还与进程调度的方式和引起进程调度的原因有关。引起进程调度的主要原因有下列几种。

(1) 正在运行的进程正常执行结束，此时需要选择另一个进程投入运行。

(2) 运行中的进程要求进行 I/O 操作后被阻塞。

(3) 进程因执行某种原语操作(如阻塞原语)而将自己阻塞，进入阻塞状态。

(4) 在执行完系统调用后，由系统态返回用户态时，进程调度程序选择一个用户进程执行。

(5) 分配给正在运行的进程的时间片已用完。

(6) 在可剥夺方式下，就绪队列中存在具有更高优先级的进程。

4. 进程调度的性能评价

进行进程调度才是真正对 CPU 进行分配和使用，因而进程的调度的优劣直接影响到 CPU 的使用效率。那么，如何衡量和评价进程调度的性能呢?

一般来说，从定量和定性两方面对进程调度的性能进行衡量和评价。

在定量方面，主要包括以下两个性能：

(1) CPU 的利用率。对 CPU 的利用率的评价可通过下列公式进行：

$$\text{CPU 的利用率} = \frac{\text{CPU 的有效利用时间}}{\text{总时间}}$$

(2) 进程在就绪队列中的等待时间与执行时间之比。

实际上，由于进程进入就绪队列的随机模型很难确定，而且进程上下文切换等也将影响进程的执行效率，从而对进程调度进行解析是很困难的。一般情况下，大多利用模拟或测试系统响应时间的方法来评价进程调度的性能。

在定性衡量方面，主要考虑以下两个方面：

(1) 可靠性。如一次进程调度是否可能引起数据结构的破坏。这要求对调度时机的选择和保存 CPU 现场十分谨慎。

(2) 简洁性。由于进程调度程序的执行涉及多个进程和必须进行上下文切换,如果调度程序过于烦琐和复杂,将会耗去较大的系统开销,加大响应时间。

4.3 调度算法

作业调度和进程调度是处理机调度中的重要内容,它们的实现算法颇为相似。它们所要解决的主要问题是选择合理的算法将处理机分配给作业或进程,以尽可能地减少处理机的空闲时间,提高系统资源的利用率。

以下调度算法有些适用于作业调度,有些适用于进程调度,有些两者都适用。

4.3.1 先来先服务调度算法

先来先服务调度算法既适用于作业调度,也适用于进程调度。

对于作业来说,先来先服务调度算法(First Come First Service,FCFS)按照后备作业进入后备队列的先后顺序排列,先进入后备队列的作业具有较高的优先级,可以优先获得调度。此时的后备队列为一个先进先出队列(First In First Out,FIFO)。

对于进程来说,先来先服务调度算法按照就绪进程进入就绪队列的先后顺序排列,先进入就绪队列的进程具有较高的优先级,可以优先获得CPU的使用权。此时的就绪队列为一个先进先出队列。

显然该算法简单,容易实现。就绪队列(后备队列)中等待时间长的进程(作业)将优先得到服务,但对于后进入就绪队列(后备队列)的短进程(作业)则需等待较长时间,故该算法没有照顾到任务的轻重缓急和特殊情况。在实际系统中,很少单独使用FCFS算法,该算法常与其他算法配合使用,例如,在优先级调度算法中,对于优先级相同的进程可以考虑采用先来先服务。

例4.1 设单道批处理系统中有4个作业J1、J2、J3和J4,其提交时间、估计运行时间及优先级如表4.1所示。采用先来先服务调度算法调度这4个作业,说明作业调度的顺序。

解: 按照先来先服务调度算法进行调度时,哪个作业先提交完毕,哪个作业就先进入后备队列,就优先被调度,由此得到作业的调度次序如表4.2所示。

表4.1 4个作业的提交情况

作业号	提交时间
J1	9:00
J2	9:50
J3	10:00
J4	10:50

表4.2 4个作业的调度情况

作业号	提交时间	调度次序
J1	9:00	1
J2	9:50	2
J3	10:00	3
J4	10:50	4

4.3.2 优先级调度算法

优先级调度算法既适用于作业调度,也适用于进程调度。

优先级调度算法(priority scheduling)指系统按照某种准则为进程(作业)确定一个优

先级，以表示该进程(作业)享有优先被调度的权利。该算法的核心是如何确定进程(作业)的优先级。这种算法可以是抢占式的，也可采用非抢占式的。

确定优先级的方法有两种：静态法和动态法。静态法根据进程的静态特性在进程(作业)执行前就确定它们的优先级，而且在执行过程中一直保持不变。动态法则不然，它将进程的静态特性和动态特性结合起来确定进程的优先级，而且随着进程的运行其优先级不断变化。

作业的静态优先级可以按其类型确定，也可以由用户根据自己的实际情况给出一个优先级，还可以按照作业要求的资源情况确定作业的优先级。优先级可作为系统确定收费的标准之一。

作业的类型一般可分为以下4种：

(1) I/O繁忙型作业；

(2) CPU繁忙型作业；

(3) I/O与CPU均衡型作业；

(4) 一般型作业。

作业一般来说没有动态优先级。

进程可以按其类型确定静态优先级，如一些系统中有用户进程和系统进程之分，一般系统进程享有比用户进程更高的优先级。也可以将进程所属作业的静态优先级直接作为进程的优先级。

基于静态优先级的进程(作业)调度算法实现简单，系统开销小。但由于进程(作业)的优先级一旦确定后即保持不变，致使调度性能不高，系统运行效率低，故现代操作系统中常采用动态优先级调度算法。

进程的动态优先级的确定过程是比较复杂的，一般主要依据以下几种原则：

(1) 根据进程占有CPU时间的长短来确定。一个进程占有CPU的时间越长，则以后被再次调度的优先级就越低；反之再次获得CPU的优先级就越高。这样短作业可以逐渐获得较高的优先级，即得到被调度的机会越来越大，可以保证它尽快执行结束。

(2) 根据进程等待CPU时间的长短来确定。一个进程在就绪队列中等待的时间越长，则它被调度选中的优先级就越高；这样先进入系统中的作业所对应的进程可以逐渐获得较高的优先级，可以保证它在有限的时间内完成。

由于进程的动态优先级随时间的推移而变化，进程调度程序要根据系统的变化情况不断地计算各进程的优先级，系统为此要付出一定的开销。

例4.2 表4.3给出了在单道程序环境下4个进程A、B、C、D进入就绪队列的时间、需要的计算时间和静态优先级(数值越大优先级越高)，系统采用基于静态优先级的非抢占式调度算法对就绪队列中的进程进行调度、分配CPU，各进程的执行顺序为A、C、D、B，执行的具体情况见表4.4。

4.3.3 轮转调度算法

轮转调度算法适用于进程调度。

轮转调度算法(round robin scheduling)是将CPU的处理时间分成固定大小的时间片，轮流将CPU分配给各个进程，从而为每个进程提供服务。如果一个进程被进程调度程序

表 4.3　进程 A、B、C、D 的情况

进入就绪队列的顺序	进程名称	进入就绪队列的时间	需要的计算时间/分	优先级
1	A	8:00	60	1
2	B	8:30	50	2
3	C	8:40	30	4
4	D	8:50	10	3

表 4.4　进程 A、B、C、D 的执行情况

进程执行顺序	进程名称	需要的计算时间/分	进程开始执行时间	进程执行结束时间
1	A	60	8:00	9:00
3	C	30	9:00	9:30
4	D	10	9:30	9:40
2	B	50	9:40	10:30

选中后用完了系统所规定的时间片而未执行结束，则它要释放 CPU 并把自己插入到就绪队列的末尾，等待下一次调度。同样位于某个阻塞队列中的进程因被唤醒而插入到就绪队列的末尾等待分配 CPU。轮转调度算法的工作原理请见图 4.3。

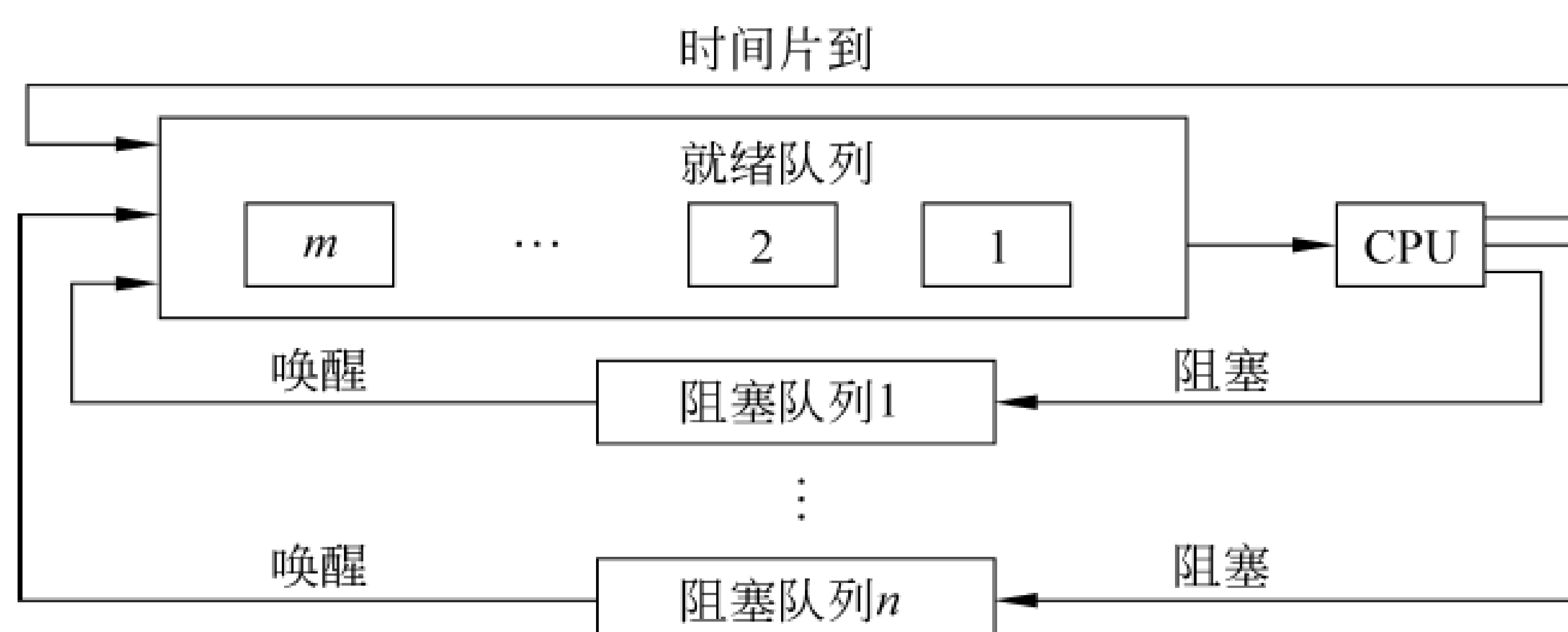


图 4.3　轮转调度算法的工作原理

轮转调度算法根据每一轮的时间片是否可变又分为固定周期轮转法和可变周期轮转法。固定周期轮转法中进程调度所使用的时间片为一个常数，保持不变；而可变周期轮转法在每一轮调度开始时，需要根据就绪队列中的进程数等参数计算该轮时间片的值，该值仅在本轮调度中有效，这样可变周期轮转法在每一轮调度中均需计算时间片的值，给系统增加了一定的时间开销。

轮转调度算法特别适合分时系统使用，如果选择合适的时间片，系统的开销不会很大。在该算法中，时间片长度的选取非常重要。时间片长度的选择会直接影响系统开销和响应时间。如果时间片长度过短，调度程序剥夺处理机的次数增多，使进程上下文切换次数大大增加，从而加重系统开销。如果时间片长度过长，比方说一个时间片能保证就绪队列中所需执行时间最长的进程能执行完毕，则轮转法变成了先来先服务法。

时间片长度的选择是根据系统对响应时间的要求 R 和就绪队列中所允许的最大进程数 N 来确定的。它可表示为

$$q=\frac{R}{N}$$

这种轮转调度算法又称为简单轮转法，其特征是就绪队列中的进程均以相同的速度向前推进。

在该调度方法中，加入到就绪队列的进程有3种情况：

(1) 分给它的时间片用完，但进程还未完成，回到就绪队列的末尾等待下次调度去继续执行。

(2) 分给该进程的时间片并未用完，只是因为请求I/O或由于进程的互斥与同步关系而被阻塞。当阻塞解除之后再回到就绪队列。

(3) 新创建进程进入就绪队列。

如果对这些进程区别对待，给予不同的优先级和时间片，可以进一步改善系统服务质量和效率。另外，如果时间片较长，则对一些需要"紧急"运行的进程和执行时间段的进程不利。可以采用下面介绍的分级轮转法来解决上述问题。

4.3.4 分级轮转调度算法

所谓分级(或多级)轮转调度算法(multilevel queue scheduling)就是根据进程性质或优先级的不同将处于就绪状态的进程组成两个或多个就绪队列，如图4.4所示。在图4.4中，每个就绪队列具有不同的优先级，其中处于高优先级就绪队列中的进程优先分配CPU；而且只有高优先级就绪队列为空时才能调度低优先级就绪队列中的进程。每个就绪队列可以采用不同的进程调度策略，而且为了体现出优先级别可以采用可抢占式调度算法。例如，系统中存在两个就绪队列，一个队列中的进程具有较高的优先级，称为前台队列，其中的各个进程所对应要完成的任务称为前台作业；另一个队列中的进程具有较低的优先级，称为后台队列，其中的各个进程所对应要完成的任务称为后台作业。

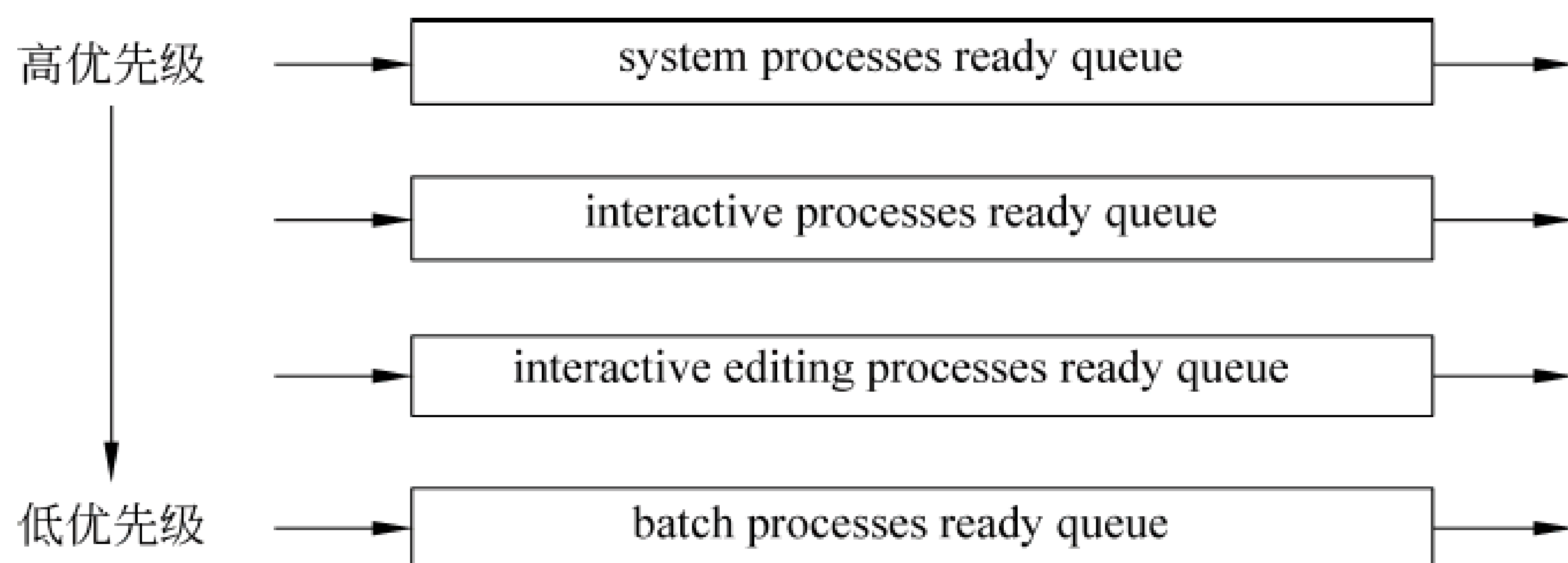


图4.4 分级轮转调度队列

一般情况下，进程调度算法将相同的时间片分配给前台就绪队列中的进程，优先满足其需求。只有前台队列中的进程全部执行完毕或因等待I/O操作而没有进程可运行时，才把CPU分配给后台队列中的进程。通常，后台就绪进程与前台就绪进程所分得的时间片并不相同，由于后台队列中的长作业进程可能长时间得不到CPU，故可以分配较长的时间片来弥补，使得长作业的进程也能得到较好的服务。例如，对于短作业的进程分配50ms的时间片，而对于长作业的进程可分配150ms的时间片，这样可以降低长作业的进程的切换次数，减少系统在切换进程上下文时的时间消耗，提高系统的执行效率。

轮转调度法适用于进程调度。

4.3.5 分级反馈轮转调度算法

在分级（或多级）轮转调度算法中，进程通常保持在一个队列中，不能在不同的队列间移动。这种方式实现简单，调度程序开销少，但不灵活。

为了克服分级轮转调度算法的缺点，提出了分级反馈轮转调度算法（multilevel feedback queue scheduling）。该算法允许进程在不同的队列间移动。该调度算法的基本思想是根据对处理器的需求及占有情况来区分进程。如果某个进程占用了太多的处理器时间，则降低该进程的优先级，并将它插入到较低优先级的进程队列中。同样，也可以将长时间等待于较低优先级队列中的进程取出，并赋予较高的优先级，然后将之插入到较高优先级的进程队列中。

现在考虑包括 n 个进程就绪队列的分级反馈轮转调度算法，如图 4.5 所示。就绪队列 1 至就绪队列 n 的优先级逐渐降低，处于高优先级就绪队列中的进程将优先得到 CPU，而每个就绪队列内均采用先来先服务（FCFS）和时间片轮转调度算法。此外，每个就绪队列将分配不同的时间片，其中优先级高的就绪队列分配较短的时间片，优先级低的就绪队列分配较长的时间片，即就绪队列 1 至就绪队列 n 的时间片是逐渐增长的。

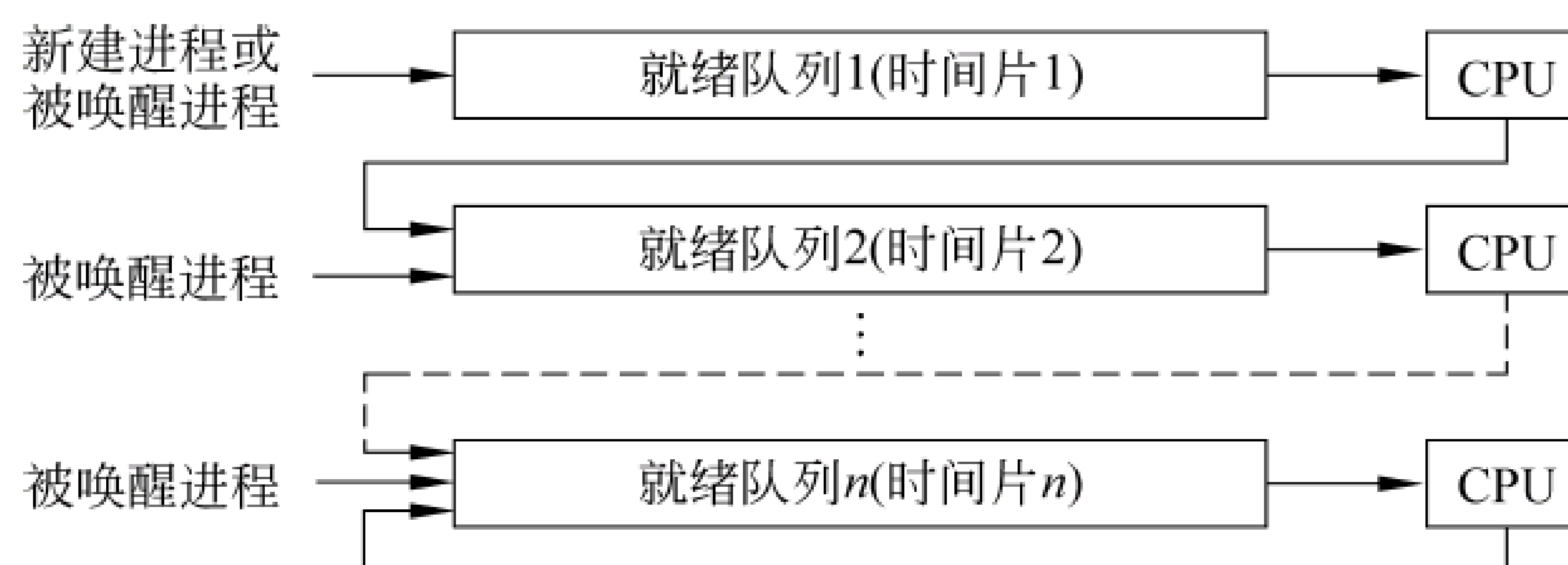


图 4.5 分级反馈轮转调度队列

各就绪队列中的进程来源分为 3 种情况：

（1）就绪队列 1 中的进程或者来源于新创建的进程，或者来源于具有相同优先级并由阻塞状态被唤醒为就绪状态的进程。

（2）就绪队列 n 中的进程一方面来源于就绪队列 $n-1$ 或者本队列中已用完时间片尚没有完全运行结束的进程，另一方面来源于具有相同优先级并由阻塞状态被唤醒为就绪状态的进程。

（3）就绪队列 $i(1<i<n)$ 中的进程一方面来源于就绪队列 $i-1$ 中刚得到 CPU 服务使用完一个时间片但尚没有完全运行结束的进程，另一方面来源于具有相同优先级并由阻塞状态被唤醒为就绪状态的进程。

当一个新进程被创建时，系统将它插入到就绪队列 1 的队尾。系统运行时，将首先选择就绪队列 1 中的进程，按照先来先服务和轮转调度的方法选择一个进程，并为之分配 CPU。若在时间片 1 内该进程没有完成其全部工作，则在时间片到后，它将被中断，让出 CPU，然后置为就绪状态并插入到就绪队列 2 的队尾。仅当就绪队列 1 为空时，才调度执行就绪队列 2 中的进程。同样，仅当序号 $i-1$ 以前的就绪队列均为空时，才能执行就绪队列 i 中的进程。

当采用可抢占式调度算法时，如果一个具有更高优先级并处于阻塞状态的进程因系统释放资源而被激活转变为就绪状态时，该进程将抢占正处于执行状态的进程所占用的处理机，优先得到服务。

由上面所述的算法可见，一个需要较短运行时间的进程只要被赋予较高的优先级，则会在很短的时间内执行完毕；而需要较长运行时间的进程只要赋予适当的优先级，也能得到很好的服务。

通常，分级反馈轮转调度算法取决于下列因素：

(1) 队列的个数。

(2) 每个队列的调度算法。

(3) 决定何时使进程升入更高优先级队列的方法。

(4) 决定何时将进程降入更低优先级队列的方法。

(5) 当进程需要服务时决定进程进入到哪一个队列中的方法。

分级反馈轮转调度算法是最通用的处理机调度算法，它可以被适当地配置以适应特定系统的需要。但该算法的确定需要考虑的因素较多，实现起来较为复杂。

分级反馈轮转法与优先级法在原理上的区别是，分级反馈轮转法中的一个进程在它执行结束之前，可能需要反复多次通过反馈循环执行，而不是优先级法中的一次执行。

4.3.6 最短作业优先调度算法

最短作业(进程)优先调度算法(Shortest Job First，SJF)是指对短作业或短进程优先进行调度并分配CPU。对于最短作业优先调度算法，是从后备作业队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存中运行。而对于最短进程优先调度算法，则是从就绪队列中选择一个估计运行时间最短的进程，将处理机分配给它。显然，这种算法使得系统单位时间内处理的作业数量达到最多，即作业周转快，系统的吞吐量大，但处理时间长的作业可能长时间得不到服务。

例4.3 设在单道程序环境下有作业1、2、3、4，其提交时间和估计运行时间如表4.5所示，请用最短作业优先算法给出作业的执行顺序。

表4.5 作业1、2、3、4的具体参数

作业号	作业提交时间	估计运行时间/分	作业号	作业提交时间	估计运行时间/分
1	8:00	120	3	9:00	10
2	8:30	30	4	9:30	15

解：当作业1提交给系统时，后备队列中只有作业1，因此马上被调度运行；当作业1运行结束时，即10点钟时，其他3个作业均已进入后备队列，按照最短作业优先算法的调度原则，先调度作业3，再调度作业4，最后调度作业2，具体执行情况如表4.6所示。

表4.6 作业执行的具体参数

作业号	作业提交时间	估计运行时间/分	开始运行时间	运行结束时间	执行顺序
1	8:00	120	8:00	10:00	1
2	8:30	30	10:25	10:55	4
3	9:00	10	10:00	10:10	2
4	9:30	15	10:10	10:25	3

4.3.7 响应比高者优先调度算法

为了克服 FCFS 和 SJF 等调度算法的缺点,既照顾短作业,又考虑作业在系统内的等待时间过长者的利益,引入了 FCFS 和 SJF 的折中算法——响应比高者优先调度算法(highest response ratio next),即以响应比作为作业调度的优先级,响应比越高,作业得到调度的优先级就越高。

响应比 R_p 的定义如下:

$$R_p = \frac{\text{等待时间} + \text{要求执行时间}}{\text{要求执行时间}} = 1 + \frac{\text{等待时间}}{\text{要求执行时间}}$$

每当系统进行调度时,均要计算每道作业的响应比 R_p,选择响应比高者优先投入运行。从 R_p 的计算公式可以看出,当一个作业要求的执行时间固定时,随着等待时间的增加,响应比则不断增大。因此,一个长作业,只要等待足够长的时间,总有机会投入运行。而对于等待时间相同的作业,要求执行时间越短,响应比越高,越会优先得到调度机会。这样既考虑了作业到达的先后顺序,也适当照顾了短作业。

这种算法的缺点是每当要调度作业时都要计算响应比,会增加一定的系统开销。

例 4.4 设在单道程序环境下有作业 1、2、3、4,其提交时间和估计运行时间如表 4.7 所示,请用响应比高者优先算法计算出每个作业被调度时的响应比,并给出作业的执行顺序。

表 4.7 作业 1、2、3、4 的具体参数

作业号	作业提交时间	估计运行时间/分	作业号	作业提交时间	估计运行时间/分
1	8:00	120	3	9:00	10
2	8:30	30	4	9:30	15

解:当作业 1 提交给系统时,后备队列中只有作业 1,因此马上被调度运行;当作业 1 运行结束时,即 10 点钟时其他 3 个作业均已进入后备队列,此时作业 2、3、4 的一些参数如表 4.8 所示。作业 3 的响应比最高,所以作业 3 是第二个被执行的作业。

表 4.8 10 点钟时作业 2、3、4 的具体参数

作业号	作业提交时间	估计运行时间/分	等待时间/分	响应比
2	8:30	30	90	4
3	9:00	10	60	7
4	9:30	15	30	3

10 点 10 分时,作业 3 执行结束,系统内还有作业 2 和作业 4。此时这两个作业的具体参数如表 4.9 所示,显然作业 2 的响应比高,故第三个运行的为作业 2。

表 4.9 10 点 10 分时作业 2、4 的具体参数

作业号	作业提交时间	估计运行时间/分	等待时间/分	响应比
2	8:30	30	100	4.33
4	9:30	15	40	3.67

10点40分时,作业2执行结束,系统内只有作业4。此时这个作业的具体参数如表4.10所示,所以作业的执行顺序为1、3、2、4。

表4.10　10点40分时作业4的具体参数

作业号	作业提交时间	估计运行时间/分	等待时间/分	响应比
4	9:30	15	70	4.67

4.4　选择调度方式和评价调度算法的若干准则

不同的处理机,调度算法有不同的特点,适合于不同的系统。在特定的应用场合选择调度算法时必须考虑各种调度算法的具体特点。

目前已有许多准则被建议用来评价处理机的调度算法。这些准则包括如下几方面。

1. CPU的利用率(CPU utilization)

CPU的利用率为CPU处于运行状态与开机运行的总时间之比。进程调度算法应该保持CPU尽可能地处于运行状态。CPU的利用率为0%～100%,在实际的系统中,CPU的利用率一般可达40%～90%。

2. 系统吞吐率(throughput)

系统吞吐率是指在单位时间内系统所能完成的作业数,它用于标识系统的处理能力。系统吞吐率是用于评价批处理系统性能的一个重要指标,因而是选择批处理作业调度策略的重要准则。系统吞吐率与批处理作业的平均长度密切相关。对于长作业,吞吐率可能是每小时一道作业;而对于短作业,吞吐率可能高达每小时10道作业。选择短作业优先算法可以获得较高的系统吞吐率。

3. 周转时间(turnaround time)

周转时间是指作业从提交给系统开始直到作业完成所经过的时间间隔,它包括作业进入内存前的等待时间、在后备队列中的等待时间、占有CPU后的运行时间以及完成各种I/O操作所需的时间,其中后3项在作业的处理过程中可能发生多次。作业i的周转时间T_i定义为

$$T_i = \mathrm{Te}_i - \mathrm{Ts}_i$$

其中Te_i是作业i的完成时间,Ts_i是作业i的提交完成时间。

作业i的周转时间T_i还可以定义为

$$T_i = \mathrm{Tw}_i + \mathrm{Tr}_i$$

其中Tw_i是作业i的等待时间(作业进入内存前的等待时间和在后备队列中的等待时间之和),Tr_i是作业i的运行时间。

周转时间常用来评价批处理系统的性能、选择作业调度方式和调度算法的重要准则之一。对于每个用户,他们希望自己作业的周转时间最短;而对于系统管理员而言,则要考虑众多用户的利益,要保证各作业的平均周转时间最短。平均周转时间T定义如下:

$$T = \frac{1}{n}\sum_{i=1}^{n} T_i$$

式中,T_i为第i个作业的周转时间,n为系统内作业的道数。

4. 带权周转时间（turnaround time with right）

为了更好地反映作业调度的性能，引入了带权周转时间。带权周转时间是指作业周转时间与作业运行时间的比。作业 i 的带权周转时间 W_i 可以定义为

$$W_i = T_i / \mathrm{Tr}_i$$

其中 T_i 是作业 i 的周转时间，Tr_i 是作业 i 的运行时间。

由于作业 i 的周转时间 T_i 还可以表示为

$$T_i = \mathrm{Tw}_i + \mathrm{Tr}_i$$

且作业 i 的带权周转时间 W_i 还可以定义为

$$W_i = (\mathrm{Tw}_i + \mathrm{Tr}_i) / \mathrm{Tr}_i$$

因此作业 i 的带权周转时间 W_i 还可表示为

$$W_i = \mathrm{Tw}_i / \mathrm{Tr}_i + 1$$

对整个系统来说，由于拥有多个作业，应计算平均带权周转时间 W，其定义如下：

$$W = \frac{1}{n}\sum_{i=1}^{n} W_i$$

还可表示为

$$W = \frac{1}{n}\sum_{i=1}^{n} \frac{T_i}{\mathrm{Tr}_i}$$

还可再表示为

$$W = \frac{1}{n}\sum_{i=1}^{n} \frac{\mathrm{Tw}_i}{\mathrm{Tr}_i} + 1$$

式中，Tr_i 是系统为第 i 个作业提供的运行时间。周转时间因包括完成各种 I/O 操作所需的时间，故通常受输入输出设备工作速度的影响较大。

5. 等待时间（waiting time）

CPU 调度算法对进程的执行时间和进行 I/O 操作的时间并没有任何影响，它仅决定一个进程在就绪队列中等待时间的长短。进程等待时间定义为一个进程在就绪队列中等待时间的总和。一个进程可能多次进入到就绪队列中等待进程调度，每次在就绪队列中等待的时间也可能不同，这主要取决于 CPU 和内存等系统资源的使用情况。

6. 响应时间（response time）

对于交互式系统而言，不是采用周转时间而常采用响应时间来评价进程调度算法的优劣。响应时间定义为一个请求提交给系统到系统响应这个请求间的时间间隔，而不是等到开始输出结果的时间间隔。它不考虑输出设备的工作速度等因素，更好地反映了进程调度算法性能的优劣。

7. 公平

公平是指所有的用户作业在提交给系统后都有均等的机会得到调度，从而避免作业提交给系统后长期得不到服务的现象发生。

8. 对资源的均衡使用

对资源的均衡使用要求调度算法调度进程或作业时应保证各类资源尽量保持相近的繁忙程度，同时要求各个同类资源也要得到均衡的使用，以保证系统的稳定性。

对于一个计算机系统，理想情况下希望得到最大的 CPU 利用率和吞吐率、最小的周转

时间和等待时间以及最快的响应时间。在大多数情况下，对这些参数进行优化以使得各项指标比较均衡。但实际上往往只能优化个别参数，而实现不了全优的目标。例如，通过最小化最大响应时间来保证用户得到理想的服务。

例 4.5 设在单道程序环境下有作业 1、2、3、4，其提交时间和估计运行时间如表 4.11 所示。按先来先服务调度算法和最短作业优先调度算法，分别计算出各作业的周转时间和带权周转时间，并计算平均周转时间和平均带权周转时间。

表 4.11 作业 1、2、3、4 的具体参数

作业号	作业提交时间	估计运行时间/分	作业号	作业提交时间	估计运行时间/分
1	8:00	120	3	9:00	10
2	8:30	30	4	9:30	15

解：(1) 按先来先服务调度算法进行调度的情况。按照先来先服务调度算法进行调度时，哪个作业先提交完毕，哪个作业就先进入后备队列，就优先被调度，由此得到作业的调度次序为 1、2、3、4。各作业的周转时间为作业的运行结束时间减去作业提交时间，带权周转时间为作业的周转时间除以作业运行时间，具体执行情况如表 4.12 所示。

表 4.12 作业执行的具体参数

作业号	作业提交时间	估计运行时间/分	开始运行时间	运行结束时间	周转时间/分	带权周转时间	执行顺序
1	8:00	120	8:00	10:00	120	1	1
2	8:30	30	10:00	10:30	120	4	2
3	9:00	10	10:30	10:40	100	10	3
4	9:30	15	10:40	10:55	85	5.67	4

平均周转时间＝(120＋120＋100＋85)÷4＝106.25(分)

平均带权周转时间＝(1＋4＋10＋5.67)÷4＝5.17

(2) 按最短作业优先调度算法进行调度的情况。当作业 1 提交给系统时，后备队列中只有作业 1，因此马上被调度运行。当作业 1 运行结束时，即 10 点钟时其他 3 个作业均已进入后备队列。按照最短作业优先算法的调度原则，先调度作业 3，再调度作业 4，最后调度作业 2。各作业的周转时间为作业的运行结束时间减去作业提交时间，带权周转时间为作业的周转时间除以作业运行时间，具体执行情况如表 4.13 所示。

表 4.13 作业执行的具体参数

作业号	作业提交时间	估计运行时间/分	开始运行时间	运行结束时间	周转时间/分	带权周转时间	执行顺序
1	8:00	120	8:00	10:00	120	1	1
2	8:30	30	10:25	10:55	145	4.83	4
3	9:00	10	10:00	10:10	70	7	2
4	9:30	15	10:10	10:25	55	3.67	3

平均周转时间＝(120＋145＋70＋55)÷4＝97.5(分)

平均带权周转时间＝(1＋4.83＋7＋3.67)÷4＝4.125

从上面的计算可以看出，采用最短作业优先调度算法时，平均周转时间和平均带权周转时间分别都小于采用先来先服务调度算法调度时的平均周转时间和平均带权周转时间。针对这组作业，采用最短作业优先调度算法，系统的得到的调度结果要好于先来先服务调度算法调度的结果。并不是所有情况下最短作业优先调度算法都好于先来先服务调度算法。

4.5 实时调度算法

实时系统广泛存在于工业、国防等领域，如工业过程控制、机器人控制、航空器/航天器控制、军事指挥控制、导弹制导和火炮随动控制等。在实时系统中操作系统，特别是进程调度算法在决定系统实时性能优劣方面起到相当重要的作用。

4.5.1 实时系统的特点

4.4 节列出了一般操作系统中常用的进程调度算法，但不同种类的操作系统所采用的进程调度算法可能存在很大差异。对于不同性质的系统，选用进程调度算法时要考虑到系统的特殊需求。

实时系统最大的特点是它的实时性，它必须对事件做出及时处理和响应，此时系统控制和处理正确与否不仅取决于计算或处理结果的正确性，而且还取决于获得结果的时间，所以此时公平性或系统的效率并不重要，但进程或线程的切换速度以及中断处理响应要快，从而保证系统具有较强的快速处理和响应能力。

实时操作系统处理的任务根据对延迟的约束要求一般可分成两类：硬实时任务和软实时任务。硬实时任务是指如果系统对任务的响应时间超出了给定的时限(deadline)，将会引起灾难性后果；而软实时任务是指允许系统对任务的响应具有一定的延迟。

实时系统的另一个特点是，根据任务发生的时间特征，实时任务又可以分为周期性任务和非周期性任务。对于不同类型的任务，操作系统可以采用不同的调度策略进行处理。对于周期性任务只要在周期内完成任务即可；而对于非周期性任务，存在一个开始时限或完成时限，在规定的时限之前开始或完成。

实时操作系统一般具有如下基本特征：

(1) 支持多线程和可抢占式调度。实时操作系统应采用可抢占式的进程/线程调度机制，给实时性要求高的进程或线程赋予较高的优先级。为了使高优先级进程能及时得到响应，它可以将正在运行的低优先级进程挂起，剥夺其处理机的使用权，强行占有处理机。但对于一些对时限要求不太严格的实时系统可以采用非抢占式调度算法，以简化调度程序和减少任务调度所花费的系统开销。

(2) 有限等待时间和响应时间。实时任务要求系统在有限的时间内给出响应并在一定的时间内处理完毕。

(3) 可靠性高，健壮性强。实时系统要求具有很高的可靠性，当出现错误时不能采用系统重启等方式来恢复系统，它要求系统在处理错误的同时仍能保证用户程序的正常运行。

(4) 操作系统的行为应该被用户所了解和掌握。一般说来，用户控制在实时系统中要

比在其他操作系统中应用得要广。对于一个典型的非实时系统，用户无法控制操作系统的调度功能，最多仅能提供一些指导，如将用户划分成多个优先级的组。但在实时系统中，用户可以控制任务的优先级。用户应能区别强、弱任务，并赋予相应的优先级。实时系统也允许用户选择进程调度算法，从而可以控制进程的执行顺序。

4.5.2 实现实时调度的基本条件

在实时系统中，硬实时任务和软实时任务都联系着一个时限。为保证系统能正常工作，实时调度必须能满足实时任务对时限的要求。因此，实现实时调度应具备下述几个条件。

1. 提供必要的信息

为了实现实时调度，系统应向调度程序提供以下信息：

（1）就绪时间。指任务成为就绪状态的时刻，对于周期性任务，它就是事先预知的一串时间序列；对于非周期任务，它可能是预知的，也可能事先不知道。

（2）开始时限和完成时限。对于典型的实时应用，只需知道开始时限或完成时限。

（3）处理时间。指一个任务从开始执行直至完成所需的时间。在有些情况下，该时间是系统提供的。

（4）资源要求。指任务执行时所需的一组资源。

（5）优先级。如果某任务的开始时限已经错过，就会引起故障，则应为该任务赋予“绝对”优先级；如果开始时限的推迟对任务的继续执行无重大影响，则可为该任务赋予“相对”优先级，供任务调度程序参考。

2. 系统处理能力强

在实时系统中，通常都有多个实时任务，实时任务的数量往往多于处理机的数量。若此时处理机的处理能力不够强，则可能出现因处理机忙不过来而使某些实时任务不能得到及时处理，从而导致发生难以预料的后果。

3. 采用抢占式调度机制

在含有硬实时任务的实时系统中，普遍采用抢占机制。当有一个更高优先级的任务到达时，允许将当前任务暂时挂起，而让高优先级的任务立即投入运行，这样可满足该硬实时任务对时限的要求。

对于一些小型实时系统，如果能预知任务的开始时限，则对实时任务的调度可采用非抢占调度机制，以简化调度程序和在任务调度时所花费的系统开销。

4. 具有快速任务切换机制

为保证要求较高的硬实时任务能及时得以运行，在实时系统中还应具有快速任务切换机制，使得任务能够快速地切换。这种机制应具有以下两方面的能力：

（1）对外部中断的快速响应能力。为使在紧迫的外部事件请求中断时系统能及时响应，要求系统具有快速硬件中断机构，还应使禁止中断的时间间隔尽量短，以免耽误其他紧迫任务的调度执行。

（2）快速的任务调度能力。为了提高调度程序进行任务切换时的速度，应使系统中的每个运行功能单位适当地小，以减少任务切换的时间开销。

4.5.3 实时调度算法的分类

可以按不同方式对实时调度算法加以分类。

根据实时任务性质的不同，可将实时调度算法分为硬实时调度算法和软实时调度算法。

按调度方式的不同，可将实时调度算法分为非抢占式调度算法和抢占式调度算法。

按因调度程序调度时间的不同，可将实时调度的算法分成静态调度算法和动态调度算法，前者是指在进程执行前，调度程序便已经决定了各进程间的执行顺序，而后者则是在进程的执行过程中，由调度程序根据情况临时决定将哪一进程投入运行。

在多处理机环境下，还可将调度算法分为集中式调度和分布式调度两种算法。

下面着重介绍按调度方式的不同对调度算法进行的分类。

1. 非抢占式调度算法

由于非抢占式调度算法比较简单，易于实现，故被广泛应用在一些小型实时系统或要求不太严格的实时控制系统中。非抢占式调度算法又可分成非抢占式轮转调度算法和非抢占式优先调度算法。

（1）非抢占式轮转调度算法

这种算法常用于工业领域的集群式控制系统中，由一台计算机控制若干个相同的（或类似的）对象，为每一个被控对象建立一个实时任务，并将它们排成一个轮转队列。调度程序每次选择队列中的第一个任务投入运行。当该任务完成后，便将它挂在轮转队列的末尾，等待下次调度运行，而调度程序再选择下一个（队首）任务运行。

这种调度算法可获得数秒至数十秒的响应时间，可用于要求不太严格的实时控制系统中。

（2）非抢占式优先调度算法

如果在实时系统中存在着对响应时间（响应时间为数百毫秒）要求较为严格的任务时，可采用非抢占式优先调度算法，为这些任务赋予较高的优先级。当这些实时任务到达时，将它们插入到就绪队列的队首，当正在执行的任务自我终止或运行完成后，才能被调度执行。

2. 抢占式调度算法

在对响应时间（响应时间为数十毫秒以下）要求较严格的实时系统中，应采用抢占式优先级调度算法。可根据抢占发生时间的不同可进一步将算法划分成以下两种。

1）基于时钟中断的抢占式优先级调度算法

在某实时任务到达后，如果该任务的优先级高于当前执行任务的优先级，这时并不立即抢占当前任务的处理机，而是等到时钟中断到来时，调度程序才剥夺当前任务的执行，将处理机分配给新到的高优先级任务。

这种调度算法能获得较好的响应效果，其调度延迟可降为几十毫秒至几毫秒。因此，此算法可用于大多数的实时系统中。

2）立即抢占的优先级调度算法

在这种调度策略中，要求操作系统具有快速响应外部事件中断的能力。一旦出现外部中断，只要当前任务未处于临界区，便立即剥夺当前任务的执行，把处理机分配给请求中断的紧迫任务。

这种算法能获得非常快的响应，可把调度延迟降低到几毫秒至 100 微秒，甚至更低。

从上述可以看出，一般情况下：

非抢占式轮转调度算法的响应时间＞非抢占式优先级调度算法的响应时间＞基于时钟中断的抢占式优先级调度算法的响应时间＞立即抢占的优先级调度算法的响应时间。

4.5.4 常用的几种实时调度算法

目前已有多种实时系统的调度算法，如时限调度算法和频率单调调度算法等。

时限调度算法是一种以满足用户要求的时限为调度原则的算法。在实时系统中的时限有两种：处理开始时限和处理结束时限。时限调度算法可以选用任一种时限作为调度的依据。时限调度算法的基本思想是：按用户的时限要求设置优先级，时限越近，要求赋予的优先级越高，应该优先占有处理机。这种算法应采用可抢占式调度方式，对于新发生的任务，如果它具有更高的优先级，则它可以剥夺正在运行进程的处理机的使用权，强行占有处理机。这种算法可以用于周期性任务和非周期性任务。

频率单调调度算法以任务发生的频率作为调度的准则，任务发生的频率越低（周期越长），其优先级越低，而发生频率高的任务则具有较高的优先级，可以优先占有处理机。这种算法可以用于周期性任务。

对于有 $m(m\geqslant 1)$ 个周期性的硬实时任务的系统，每个任务的处理时间表示为 C_i，周期时间表示为 T_i，但在处理器系统中，使用频率单调调度算法时的充分条件是

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\cdots+\frac{C_m}{T_m}\leqslant m(2^{\frac{1}{m}}-1)$$

使用频率单调调度算法时的必要条件是

$$C_i\leqslant T_i$$

例 4.6 设系统中有两个周期性任务，任务 A 的周期时间为 20ms，每个周期的处理时间为 10ms；任务 B 的周期时间为 50ms，每个周期的处理时间为 25ms。利用时限调度算法进行调度，按处理结束时限设置优先级，即距处理结束时限越近，优先级越高。说明系统对这两个任务的调度情况。

解：这两个任务的部分预计发生、执行和结束时限如表 4.14 所示。

表 4.14 两个任务的部分预计发生、执行和结束时限

进程	发生时限/ms	处理时间/ms	结束时限/ms	进程	发生时限/ms	处理时间/ms	结束时限/ms
A(1)	0	10	20	⋮	⋮	⋮	⋮
A(2)	20	10	40	B(1)	0	25	50
A(3)	40	10	60	B(2)	50	25	100
A(4)	60	10	80	B(3)	100	25	150
A(5)	80	10	100	B(4)	150	25	200
A(6)	100	10	120	⋮	⋮	⋮	⋮
A(7)	120	10	140				

任务 A 和任务 B 的调度顺序和相对时间如图 4.6 所示。

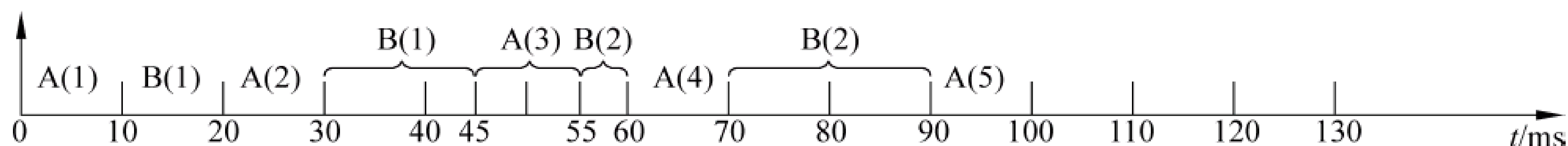


图 4.6 任务 A 和任务 B 的调度顺序和相对时间

如图 4.6 所示，在开始时，进程 A(1)和进程 B(1)的结束时限比较结果，进程 A(1)的结束时限最近，从而调度进程 A(1)执行。进程 A(1)的实际结束时间为 10ms，小于 20ms 的时限要求。紧接着，进程 B(1)被调度执行。在执行到时间为 20ms 时，进程 A(2)进入就绪状态。由于进程 A(2)的结束时限为 40ms，早于进程 B(1)的结束时限 50ms，从而进程 B(1)占有的 CPU 被进程 A(2)抢夺。进程 A(2)的实际结束时间为 30ms，小于要求时限 40ms。在进程 A(2)结束之后，进程 B(1)再次占有 CPU 继续执行，当进程 B(1)执行到时间为 40ms 时，进程 A(3)进入就绪状态。但是，由于进程 A(3)的结束时限为 60ms，远于进程 B(1)的结束时限 50ms，从而进程 B(1)继续执行。到 45ms 时进程 B(1)执行结束，进程 A(3)开始执行。进程 A(3)执行到 50ms 时，进程 B(2)进入就绪状态。由于进程 A(3)的结束时限为 60ms，早于进程 B(2)的结束时限 100ms，进程 A(3)继续执行，到 55ms 时进程 A(3)执行结束。由于此时只有进程 B(2)为就绪状态，故进程 B(2)开始执行。到 60ms 时，进程 A(4)进入就绪状态。由于进程 A(4)的结束时限为 80ms，早于进程 B(2)的结束时限 100ms，从而进程 B(2)占有的 CPU 被进程 A(4)抢夺。进程 A(4)的实际结束时间为 70ms，小于要求时限 80ms。在进程 A(4)结束之后，进程 B(2)再次占有 CPU 继续执行，当进程 B(2)执行到时间为 80ms 时，进程 A(5)进入就绪状态。由于进程 B(2)和进程 A(5)的结束时限都是 100ms，而此时进程 B(2)占有 CPU，如果进程 B(2)继续执行不用进行切换，如果让进程 A(5)执行的话就要进行切换，故让进程 B(2)继续执行到 90ms 结束，再让进程 A(5)执行。

时限调度算法不但可以用于周期性任务调度，也可以用于非周期性任务调度。

例 4.7 针对上例中的两个任务能否采用频率单调调度算法进行调度？说明理由。

解：因为

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}=\frac{10}{20}+\frac{25}{50}=1$$

而

$$m(2^{\frac{1}{m}}-1)=2(2^{\frac{1}{2}}-1)=0.828$$

即

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}\geqslant 2(2^{\frac{1}{2}}-1)$$

采用频率单调调度算法的充分条件是

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}\leqslant 2(2^{\frac{1}{2}}-1)$$

所以这两个任务不能采用频率单调调度算法进行调度。

4.6 Linux 的进程调度

在 Linux 中，进程是一个非常重要的概念，它是资源分配的基本单位，运行于自己的虚拟地址空间中，多个进程可以并发执行。同时进程也是 Linux 内核的调度单位，当进程调度时机成熟时，进程调度程序在多个进程间进行合理选择，为条件最佳的进程分配处理机并使之投入运行。

Linux 中的内核线程采取了与进程一样的表示和管理方式，Linux 使用进程调度统一处理进程和内核线程，所以通过进程调度就可以得知线程调度的具体情况。

一般来讲，进程调度主要包括两个方面：进程调度的时机和调度算法。下面分别论述 Linux 系统在这两方面的具体实现方式。

4.6.1 调度的时机

调度的时机指何时重新进行进程调度，即何时重新分配 CPU。Linux 进程调度的时机和现代操作系统进程调度的时机基本一致。Linux 中设置了进程调度标志 need_resched，当该标志为 1 时，可以执行进程调度程序。当进程调度时机到来时，内核通过检测进程调度标志以决定是否执行进程调度程序。通常，引起 Linux 系统中进程调度的原因有如下几种：

（1）CPU 执行的进程发生状态转换，如进程终止、进程睡眠等。进程在执行过程中调用 sleep()、exit()或 wait()等函数将引起其状态转变，这些函数的执行会调用调度程序进行进程调度。

（2）就绪队列中增加了新进程。

（3）正在执行的进程所分配的时间片用完。

（4）执行系统调用的进程返回到用户态。

（5）系统内核结束中断处理返回到用户态。

（6）直接执行调度程序。

4.6.2 进程调度算法

当进程的调度时机满足后系统就要发生进程调度。Linux 的进程调度是将优先级调度、轮转法调度、先来先服务调度以及多级反馈轮转调度综合起来的一种高效调度算法。如前所述，Linux 中的进程分为实时进程和普通进程。实时进程要求响应的速度快而且可靠性要高，因此应比普通进程具有更高的调度优先级。同时，Linux 也充分考虑到了各种进程调度的公平性，针对不同类型的进程采用了不同的调度策略。对于实时进程采用了基于优先级的轮转调度算法和基于优先级的先来先服务调度算法，而对于普通进程则采用了基于优先级的轮转调度算法。

3.6.1 节介绍了 Linux 的进程控制块：task_struct 中的 4 个成员：policy、priority、rt_priority 和 counter。其中 unsigned long policy 的值表示不同类型进程的调度策略，其取值范围为{SCHED_OTHER，SCHED_FIFO，SCHED_RR}，分别对应普通进程优先级轮转调度算法、实时进程基于优先级的先来先服务调度算法及基于优先级的轮转调度算法。可见 3 种调度算法均是基于优先级的，普通进程的优先级由 priority 确定，而实时进程的优先级由 rt_priority 确定。counter 用于指出轮转法中时间片的大小，其初值分别为 priority 和 rt_priority。进程启动后 counter 值随时钟周期递减，当减至零时将引起进程调度，从而保证当前进程不会一直占有处理机。

Linux 确定进程调度策略时综合考虑各种因素，并将它们集中反映到进程的优先级上。进程的调度策略均以优先级为依据，各类进程的优先级计算如下：

实时进程的优先级＝1000＋rt_priority

普通进程的优先级＝counter

其中 counter 的初值为 priority。

由上面进程的优先级计算公式可以看出，对于轮转调度算法，各进程时间片的初始值取自它们的优先级，这样将进程的时间片和优先级很好地对应起来，使得优先级越高的进程得到处理机服务的时间就越长。

在 Linux 系统中，仅存在一个就绪进程队列。实时进程和普通进程均按照进入就绪队列的先后顺序排列，各就绪进程通过 PCB 中就绪队列双向链表的前、后向指针（struct task_struct * next_run，* prev_run）链接起来。在发生进程调度时，进程调度程序从就绪队列中选择优先级最高的进程占有处理机并执行。所以 Linux 进程调度的基本策略是优先级调度。由进程优先级的计算公式可以看出实时进程的优先级通常高于普通进程，这样使得处于同一就绪队列中的进程按照优先级的不同实际上分成了两个队列。显然，实时进程得以优先调度。

对于实时进程，Linux 采用基于优先级的先来先服务算法和基于优先级的时间片轮转算法，具体采用哪种算法由其 PCB 中的 policy 域来决定。当采用基于优先级的轮转法时，各进程按照其优先级和在就绪队列中的顺序轮流地接受一个时间片长的处理机服务，此时优先级高的进程应得到优先调度。当某个进程的时间片用完后马上重新分配新的时间片并插入到就绪队列，而且保持其优先级不变，然后等待再次被调度。这种情况下，所有实时进程实际上是按照优先级分成了多级队列，优先级相同的进程按照进入就绪队列的时间顺序排列在同一个就绪队列中。轮转算法优先调度优先级最高的就绪队列中的进程，只有高优先级别的就绪队列为空时才调度位于低优先级别就绪队列中的进程。可见，在这种调度方式下，进程的优先级越高，响应越及时。又因为采用轮转算法，故具有相同优先级的进程和具有不同优先级的进程根据其轻重缓急均能得到相应的服务。

当实时进程调度发生时，进程调用程序调用可运行度量函数 goodness()，此函数将依据实时优先级 rt_priority 等参数来衡量哪一个就绪进程最值得调度。此时 counter 的值只用来表示剩余时间片的多少，不作为进程调度的依据。

普通进程的调度采用基于优先级的轮转调度算法，进程调度的依据是 PCB 中 priority 值的大小。这类进程在创建时便给其 PCB 中的 priority 赋一个初值，它在进程运行过程中一直保持不变。这个值同时也作为 counter 的初值，但 counter 的值在进程的运行过程中不断减少以表示剩余时间片的多少。由此可见，priority 的初值实际上就是分配给该进程的初始时间片。那么一个进程的优先级越高，它最初将得到的服务时间就越长。在进程执行过程中，其 counter 值逐渐减少到 0 时，表示该进程已用完了所有的时间片，此时应放弃 CPU，然后插入到就绪队列的末尾，但不马上为其分配时间片，需等待就绪队列中已分配时间片的进程均用完各自的时间片后，才重新为每个进程分配新的时间片，然后进行新一轮调度。这时 counter 应重新被赋值，以使得普通进程有机会被重新调度。显然，当某进程的 counter 值减为 0 时，会完全放弃对 CPU 的使用，其他进程运行的机会就会增加，所以此时采用的进程调度算法也称为动态优先级法。分配给每个进程的时间片在进程创建时可由系统设定为默认值，也可由用户通过系统调用来设定。

总之，Linux 中的进程调度以进程的优先级为统一的调度依据，调度算法所使用的数据结构简单，并将多种调度策略有机地结合起来，同时兼顾各类进程的特点。对于实时性要求高的进程，采用基于优先级的先来先服务调度策略，保证用最快的速度响应；而对于实时性

要求较低的进程，可以采用基于优先级的轮转调度算法，保证各个进程同时获得较快的响应，从而实现对所有进程的公平、合理和高效调度。

4.7 本章小结

处理机是计算机中最宝贵的系统资源，处理机利用率的高低直接关系到系统的使用效率和资源利用率。操作系统一般采用分级调度的方式对处理机进行管理，分级调度包括作业调度、交换调度、进程调度和线程调度。现代操作系统中处理机的有效使用与否主要取决于进程调度。可以说，进程调度是操作系统的核心，进程调度算法的优劣直接关系到计算机系统的运行效率。

调度算法有多种，如先来先服务调度算法、优先级调度算法、轮转调度算法、分级轮转调度算法、分级反馈轮转调度算法、最短作业优先调度算法和响应比高者优先调度算法。不同的操作系统可以选用不同的调度算法或它们的组合。调度算法的选择可以依据多个方面，如公平性、高效性、响应时间和周转时间等。

实时系统有着与一般系统不同的特点，其最重要的特性是实时性和可靠性。实时系统的操作系统也与一般的操作系统不同，采用的调度策略和调度方法就与一般的操作系统不同，实时系统的操作系统常用的调度算法有时限调度算法和频率单调调度算法，时限调度算法既适用于周期性任务的调度，也适用于非周期性任务的调度；而频率单调调度算法只适用于周期性任务的调度，并且是有条件的。

Linux 操作系统是一个有效进行进程调度的范例，它较好地考虑到了进程的各种特性，将多种进程调度策略有机地结合起来，实现了对进程的公平、高效调度。

习　　题

1. 为什么说处理机调度是现代操作系统的核心？
2. 简单说明操作系统中分级调度的主要内容。
3. 什么是作业调度和进程调度？
4. 作业调度、进程调度和线程调度各存在于哪些操作系统中？
5. 在多级调度过程中，哪一级调度发生的频率最高？
6. 中级调度的目的什么？需要额外的代价吗？
7. 说明作业调度的主要功能。
8. 说明进程调度的主要功能。
9. 进程调度主要有哪几种方式？
10. 举例说明引起进程调度的原因。
11. 理解进程调度中静态优先级和动态优先级的概念以及它们的确定方法。
12. 为什么说分级反馈轮转调度算法能够较好地满足各种用户的要求？
13. 确定分级反馈轮转调度算法时所要考虑的因素有哪些？
14. 说明响应比高者优先调度算法的实现原理及其优点。
15. 假设有4个作业，它们的提交时间和需要的计算时间如表4.15所示。这些作业在

一台处理机上按单道方式运行，若采用下列 4 种调度算法进行调度：

（1）先来先服务调度算法；

（2）短作业优先调度算法；

（3）非抢占式的静态优先级调度算法（数值大者优先级高）；

（4）响应比高者优先调度算法。

请分别给出这些作业的执行顺序、每个作业的周转时间和带权周转时间，以及平均周转时间和平均带权周转时间。

表 4.15　4 个作业的情况

作业	作业提交时间	所需的计算时间/分	优先级
J1	10:00	90	1
J2	10:10	40	3
J3	10:30	20	2
J4	11:00	10	4

16. 某作业 9:00 到达，预计运行时间为 2 个小时，13:00 该作业开始运行，请计算该作业的响应比。

17. 假设有 4 个作业，它们的提交时间和需要的计算时间如表 4.16 所示。这些作业在一台处理机上按单道方式运行，如采用响应比高者优先调度算法，请给出这些作业的执行顺序。

表 4.16　4 个作业的提交时间和需要的计算时间

作业	作业提交时间	所需的计算时间/分	作业	作业提交时间	所需的计算时间/分
J1	8:00	120	J3	9:00	20
J2	8:30	60	J4	9:10	40

18. 在单处理机系统中的进程就绪队列和阻塞队列最多只能有一个吗？

19. 简单说明实时操作系统所处理任务的种类及其具体含义。

20. 实时操作系统选择进程调度算法时应考虑的主要问题是什么？

21. 评价作业或进程调度算法优劣的主要指标有哪些？

22. 设有 4 个作业 J1、J2、J3、J4，它们同时到达，需要的运行时间分别为 T_1、T_2、T_3、T_4，且 $T_4 \geqslant T_3 \geqslant T_2 \geqslant T_1$，若它们在一台处理机上按单道方式运行，采用最短作业优先算法，则平均作业周转时间是多少？

23. 有 5 个待执行的作业，分别是 A、B、C、D、E，各自估计的运行时间是 9、6、3、5、x，试问采用哪种运行次序平均周转时间最短，其平均周转时间是多少？

24. 设周期性任务 P1、P2、P3 的周期 T_1、T_2、T_3 分别为 100ms、150ms、350ms，处理时间分别是 20ms、40ms、100ms，请问能否采用频率单调调度算法对它们进行调度？说明原因。

25. 现有 3 个同时到达的作业 J1、J2、J3，它们的执行时间分别为 T_1、T_2、T_3，且 $T_1<$

$T_2 < T_3$，系统按单道方式运行且采用短作业优先算法，则平均周转时间是多少？

26. 设系统中从两个不同的数据源DA和DB周期性地采集数据任务，其中DA的周期时间为30ms，每个周期的处理时间为15ms；DB的周期时间为75ms，每个周期的处理时间为38ms。利用时限调度算法进行调度，按处理结束时限设置优先级，即处理结束时限越近，优先级越高，请利用图和表说明系统对这两个任务的调度情况。

27. 在单CPU和两台输入输出设备(I1、I2)的多道程序设计环境下，同时投入3个作业Jobl、Job2、Job3运行。这3个作业对CPU和输入输出设备的使用顺序和时间如下所示：

Job1：I2(30ms)，CPU(10ms)，I1(30ms)，CPU(10ms)，I2(20ms)

Job2：I1(20ms)，CPU(20ms)，I2(40ms)

Job3：CPU(30ms)，I1(20ms)，CPU(10ms)，I1(10ms)

假定CPU、I1、I2都能并行工作，Job1优先级最高，Job2次之，Job3优先级最低，优先级高的作业可以抢占优先级低的作业的CPU，但不抢占I1和I2。试求：

(1) 3个作业从投入到完成分别需要的时间。

(2) 从投入到完成的CPU利用率。

(3) I/O设备利用率。

28. 请说明实时调度算法中时限调度算法和频率单调调度算法的适用范围。

29. Linux系统的PCB中与进程调度有关的几个参数是什么？具体代表什么含义？

30. 理解说明Linux系统中就绪进程队列的组织方式。

31. 说明Linux系统中引起进程调度的原因。

32. 简述Linux系统中实时进程的调度过程。

33. 简述Linux系统中普通进程的调度过程。

第 5 章　存 储 管 理

存储器是存储程序和数据等信息的载体，是计算机的 5 个组成部分之一，也是最重要的系统资源。正在运行的程序和数据以及各种控制用的数据结构都必须占用一定的存储空间，因此，存储管理的效果直接影响到系统性能。

存储器由内存（primary storage，也称为主存）和外存（secondary storage）组成。内存存取速度快，价格较高，一般容量有限；而外存存取速度较慢，价格便宜，可以达到海量存储。众所周知，将要运行的程序必须装入内存才能有机会分配 CPU 而得到执行，因而内存空间的大小就限制了所要装入程序的长度和多道程序环境下程序道数。操作系统的存储管理采用一定的措施将内存和外存有机地结合起来，使得实际运行的程序不受内存空间大小的限制，而且可以做到多道程序同时运行。

本章讨论内存管理中所采用的技术，主要包括常用的内存管理方法以及相应所采用的内存分配和回收算法、内外存地址转换、内存数据共享与保护、内存扩充、覆盖技术和交换技术等。而有关外存管理方面的内容将在第 7 章中讨论。

5.1　存储管理的功能

存储管理完成的主要功能有内存的分配和回收、地址转换、内存数据共享与保护和内存扩充等。

5.1.1　内存的分配与回收

内存的分配与回收是内存管理的主要功能之一。无论采用哪一种管理和控制方式，能否把外存中的数据和程序调入内存，取决于能否在内存中为它们安排合适的位置和空间。因此，存储管理要为每一个作业或进程分配内存空间（称为内存分配）。另外，当作业或进程结束之后，存储管理模块又要及时回收作业或进程所占用的内存资源（称为回收或去配），以便给其他作业或进程运行时使用。

为了有效合理地利用内存，在设计内存的分配和回收方法时，必须考虑和确定以下数据结构和策略：

（1）分配结构。供分配程序使用的数据结构，登记内存使用情况。例如内存空闲区表和空闲区队列等。

（2）放置策略。确定调入内存的程序和数据应放在内存中的具体位置。这是一种选择内存空闲区的策略。

（3）交换策略。在需要将某个程序段和数据调入内存时，如果内存中没有足够的空闲区，此时应由交换策略来确定把内存中的哪些程序段和数据段调出内存，以便腾出足够的空间来装载要调入的程序或数据。

（4）调入策略。外存中的程序段和数据段在什么时间按什么样的控制方式调入内存。

调入策略与内外存数据流动控制方式有关。常用的控制方式有交换方式、请求调入方式和预调入方式。

(5) 回收策略。如何把运行结束的作业或进程所用的内存空间收回。回收策略包括两点：一是回收的时机；二是对所回收的内存空闲区和已存在的内存空闲区的调整，如相邻空闲存储块的合并。

5.1.2 地址转换

用户源程序经过编译或汇编后形成的目标代码中出现的地址通常为相对地址，即规定目标程序的首地址为零，而其他指令中的地址都是相对于首地址而定的，这种地址通常称为逻辑地址，有时也称为虚拟地址。把逻辑地址组成的空间称为虚拟存储器(virtual storage 或 virtual memory)，也称虚拟空间。

主存储器中各存储单元的编号称为物理地址。物理地址有时也称为绝对地址。就系统而言，其主存的全部物理单元的集合称为物理存储空间。

处理器执行指令时是按物理地址进行的，所以，在作业调度选中某一用户作业，将其程序或数据放入主存时，必须把该用户作业地址空间中的逻辑地址转换成主存中的物理地址，这样才能得到信息在主存中的真实存放处，这个过程称为地址转换，也称为地址重定位或地址映射。

地址重定位就是要建立逻辑地址与物理地址间的对应关系。实现地址重定位或地址映射的方法有两种：静态地址重定位和动态地址重定位。

1. 静态地址重定位

静态地址重定位(static address relocation)是在程序执行之前由装配程序完成地址转换工作。显然，对于虚拟空间内的指令或数据来说，静态重定位只完成一个首地址不同的连续地址转换。它要求所有待执行的程序和所处理的数据必须在程序执行之前完成它们之间的链接，否则将无法得到正确的内存地址和内存空间。

静态重定位的优点是不需要硬件支持。但是，使用静态重定位方法进行地址转换无法实现虚拟存储器。这是因为，虚拟存储器呈现在用户面前的是一个在物理上只受内存和外存总容量限制的存储系统，这要求存储管理系统只把进程执行时频繁使用和立即需要的指令与数据等存放在内存中，而把那些暂时不需要的部分存放在外存中，待需要时自动调入，以提高内存的利用率和并行执行的作业道数。显然，这是与静态重定位方法相矛盾的，静态重定位方法一旦将程序装入内存之后就不能再移动，并且必须在程序执行之前将有关部分全部装入内存。

静态重定位的另一个缺点是必须占用连续的内存空间，这就难以做到程序和数据的共享。

2. 动态地址重定位

动态地址重定位(dynamic address relocation)是在程序执行过程中，在 CPU 访问内存之前，将要访问的程序或数据地址转换成内存地址。动态重定位需要硬件的支持。

动态重定位机构需要一个(或多个)基地址寄存器(BR)和一个(或多个)程序逻辑地址寄存器(VR)。指令或数据的内存地址 MA 与逻辑地址的关系为

$$MA=(BR)+(VR)$$

这里，(BR)与(VR)分别表示寄存器 BR 与 VR 中的内容。

动态重定位的主要优点如下：

(1) 可以对内存进行非连续分配。对于同一进程的各分散程序段，只要把各程序段在内存中的首地址统一存放在不同的 BR 中，则可以由地址变换机构转换得到正确的内存地址。

(2) 动态重定位提供了实现虚拟存储器的基础。因为动态重定位不要求在作业执行前为所有程序分配内存，也就是说，可以部分地、动态地分配内存。从而，可以在动态重定位的基础上，在执行期间采用请求方式(或预调入方式)为那些不在内存中的程序段分配内存，以达到内存扩充的目的。

(3) 组成作业的各程序段可以分散存放在不同的内存区域，有利于程序段的共享。

5.1.3 内存信息的共享与保护

内存信息的共享与保护也是内存管理的重要功能之一。在多道程序设计环境下，内存中的许多用户或系统程序和数据段可供不同的用户进程使用，称为共享。这种资源共享将会提高内存的利用率。除了被允许共享的部分之外，要限制各进程只在自己的存储区中活动，各进程不能对别的进程的程序和数据段产生干扰和破坏，因此须对内存中的程序和数据段采取保护措施。

常用的内存信息保护方法有硬件法、软件法和软硬件结合法。

上下界保护法是一种常用的硬件保护法。上下界存储保护技术要求为每个运行着的进程或数据段设置一对上下界寄存器。上下界寄存器中装有被保护程序段或数据段的起始地址和终止地址。在程序执行过程中，在对内存进行访问操作时首先进行访址合法性检查，即检查经过重定位后的内存地址是否在上、下界寄存器所规定的范围之内。若在规定的范围之内，则访问是合法的；否则是非法的，并产生访址越界中断。

保护键法也是一种常用的存储保护法。保护键法为每一个被保护存储块分配一个单独的保护键。在程序状态字中则设置相应的保护键开关字段，对不同的进程赋予不同的开关代码，并与被保护的存储块中的保护键匹配。保护键可设置成对读写同时保护的或只对读、写进行单项保护的。例如，图 5.1 中的保护键 0 就是对 2KB 到 4KB 的存储区进行读写同时保护的，而保护键 2 则只对 4KB 到 6KB 的存储区进行写保护。如果进程的开关代码与保护键匹配或存储块未受到保护，则访问该存储块是允许的，否则将产生访问出错中断。

2KB	0	R	W
4KB	2	W	
6KB			

图 5.1 保护键法示意图

软硬件法结合是界限寄存器与 CPU 的用户态或核心态工作方式相结合的保护方式。在这种保护模式下，用户态进程只能访问那些在界限寄存器所规定范围内的内存部分，而核心态进程则可以访问整个内存地址空间。

5.1.4 内存的扩充

主存容量是有限的，当主存资源不能满足用户作业需求时，例如当有一个比主存容量还要大的作业要运行时，或为使多个作业在主存中并发运行时，就需要由操作系统利用辅助存储器对主存进行扩充，这个过程对用户作业是透明的(即用户感知不到)。这里所说的扩充

是指使用存储管理软件来实现内存在逻辑上的扩充，而不是利用硬件实现的扩充。通过这种内存扩充方式可以实现系统运行的作业大小只受内存容量和外存容量之和的限制，而不是受内存大小的限制，从而系统能够运行的作业大小得以增大。

一般将进程的程序段、数据段等虚拟地址组成的虚拟空间称为虚拟存储器（virtual storage 或 virtual memory）。虚拟存储器只规定每个进程中互相关联信息的相对位置，每个进程都拥有自己的虚拟存储器，且虚拟存储器的容量是由计算机的地址结构和寻址方式确定的。虚拟存储器到物理存储器的变换是操作系统必须解决的问题。要实现这个变换，必须要有相应的硬件支持，并使这些硬件能够完成统一管理内存和外存之间数据和程序段自动交换的虚拟存储器功能。虚拟存储器的大小只受内存容量和外存容量之和的限制，其常见的实现方式有动态页式、动态段式和动态段页式存储管理方法，故这 3 种管理方法也称为虚拟存储管理方法，在后面详细介绍。

5.2 覆盖和交换技术

覆盖技术和交换技术是在多道环境下用来扩充内存的两种方法。覆盖技术主要用在早期的操作系统中，而交换技术则在现代操作系统中仍具有较强的生命力。

5.2.1 覆盖技术

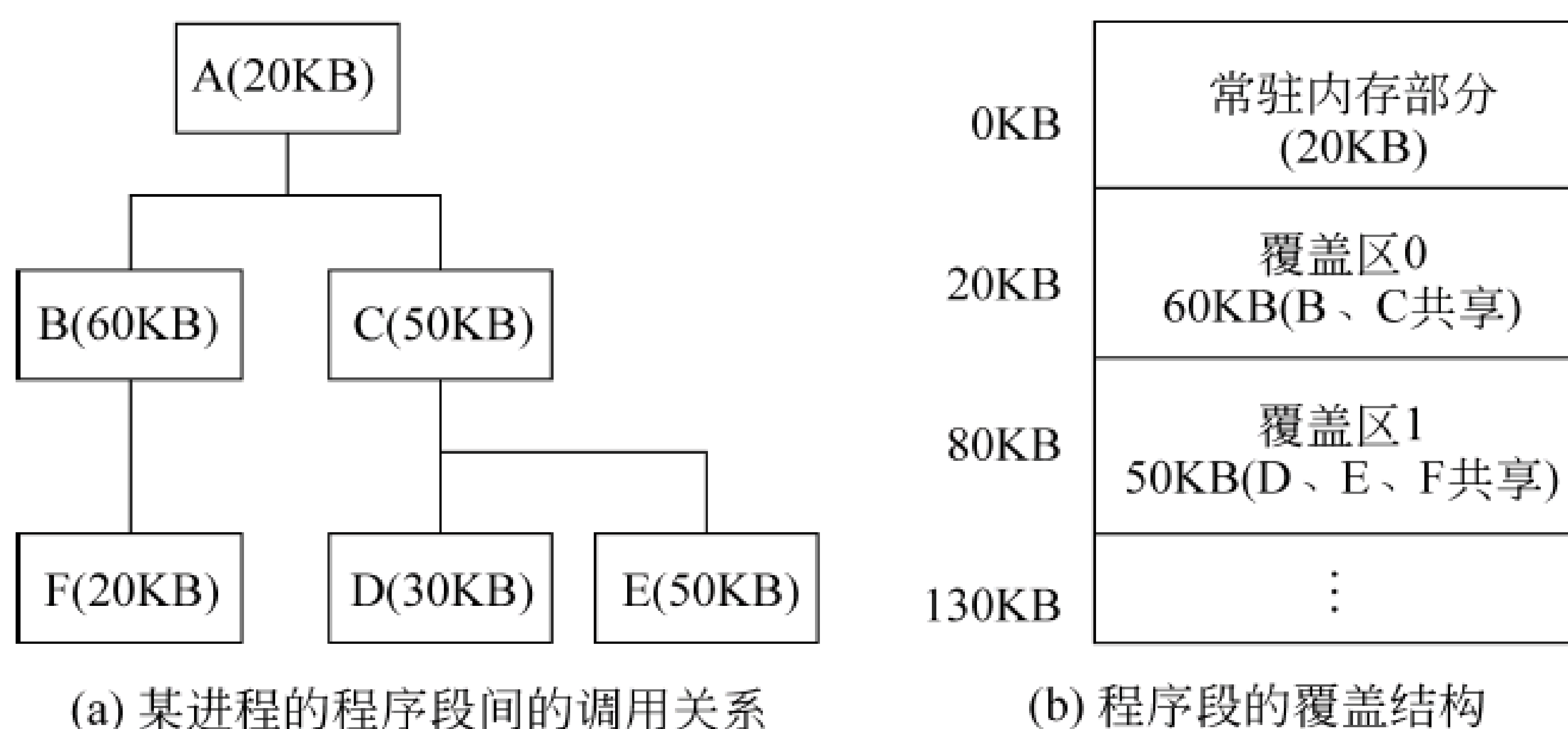
一般来说，程序具有两个特点：第一，程序执行时有些部分是彼此互斥的，即在程序的一次执行中，执行了这部分就不会去执行另一部分。第二，程序的执行往往具有局部性，在一段时间里可能循环执行某些指令或多次访问某一部分的数据。所以，即使把作业有关的信息全部装入主存储器，在实际执行时也不会同时使用这些信息，甚至有些信息在作业执行的整个过程中都不会被使用。可见，没有必要把作业的全部信息同时存放在主存储器中。在装入部分信息的情况下，只要调度得好，完全可以保证作业的正确执行。

覆盖技术正是基于程序的这两个特点提出来的。其基本思想是：把程序划分为若干个功能上相对独立的程序段，按照程序的逻辑结构让那些不会同时执行的程序段共享同一块内存区。通常，这些程序段都被保存在外存中，当有关程序段的先头程序段已经执行结束后，再把后续程序段调入内存覆盖前面的程序段。这使得用户看来好像内存扩大了，从而达到了内存扩充的目的。

覆盖技术要求程序员提供一个清楚的覆盖结构。即程序员必须完成把一个程序划分成不同的程序段，并规定好它们的执行和覆盖顺序，写成一个覆盖描述文件随同作业一起交给操作系统。操作系统根据程序员提供的覆盖描述文件来完成程序段之间的覆盖。一般来说，一个程序究竟可以划分为多少段，以及让其中的哪些程序段共享哪一内存区只有程序员清楚。这要求程序员既要清楚地了解程序所属进程的虚拟空间及各程序段所在虚拟空间的位置，又要求程序员懂得系统和内存的内部结构与地址划分，因此，程序员负担较重。所以，覆盖技术大多由对操作系统的虚拟空间和内部结构很熟悉的程序员来使用。

例如，设某进程的程序正文段由 A、B、C、D、E 和 F 共 6 个程序段组成。它们之间的调用关系如图 5.2(a)所示，从图中可以看出，程序段 B 不会调用 C，程序段 C 也不会调用 B。因此，程序段 B 和 C 无须同时驻留在内存，它们可以共享同一内存区。同理，程序段 D、E、

F 也可共享同一内存区。其覆盖结构如图 5.2(b)所示。



(a) 某进程的程序段间的调用关系　　(b) 程序段的覆盖结构

图 5.2　覆盖示例

在图 5.2(b)中,整个程序正文段被分为两个部分。一个是常驻内存部分,该部分与所有的被调用程序段有关,因而不能被覆盖,该区域被称为驻留区,这一部分中的程序称为根程序。显然,程序段 A 是根程序,所占的部分为驻留区。另一部分是覆盖部分,分为两个覆盖区。其中,一个覆盖区由程序段 B、C 共享,其大小为 B、C 中所需容量最大者,其容量为 50KB;另一个覆盖区为程序段 F、D、E 共享,其容量为 50KB。不采用覆盖技术时该程序正文段所需内存空间是 230KB,采用了覆盖技术只需 130KB 的内存空间即可开始执行。

5.2.2　交换技术

在多道程序环境或分时系统中,多个作业或进程同时执行。但是,这些同时存在于内存中的作业或进程,有的处于执行状态或就绪状态,而有的则处于等待状态。通常等待时间比较长,例如从外存软磁盘读一块数据到内存有时要花 0.1 秒到 1 秒左右的时间。如果让这些等待中的进程继续驻留内存,将会造成存储空间的浪费。因此,应该把处于等待状态的进程换出内存。

实现上述目标的常用方法之一就是交换(swap)。交换指先将内存某部分的程序或数据写入外存交换区,再从外存交换区中调入指定的程序或数据到内存中来。

交换进程由换出和换入两个过程组成,其中换出(swap out)过程把内存中的数据或程序换到外存交换区,而换入(swap in)过程把外存交换区中的数据或程序换到内存分区中。

交换技术大多用在小型机或微机系统中。这样的系统大部分采用固定的或可变分区方式管理内存。

交换主要是在不同的进程或作业之间进行,而覆盖则主要在同一个作业或进程内进行。另外,覆盖只能在那些无关的程序段之间进行。

5.3　分区存储管理

分区存储管理是把主存储器中的用户区作为一个连续区或分成若干个连续区进行管理,当划分多个连续区时,可采用固定分区方式或可变分区方式进行管理。由此可见,分区存储管理可分为单分区管理和多分区管理,多分区管理又分为固定分区管理和可变分区管理。

5.3.1 单分区存储管理

单分区存储管理是一种最简单的存储管理方式。在这种管理方式下,用户区域作为一个连续的分区分配给一个作业使用,即在任何时刻主存储器中只有一个作业。单分区的存储管理只适用于单用户的情况,个人计算机和专用计算机系统可采用这种存储管理方式。图 5.3 是单分区存储管理示意图。

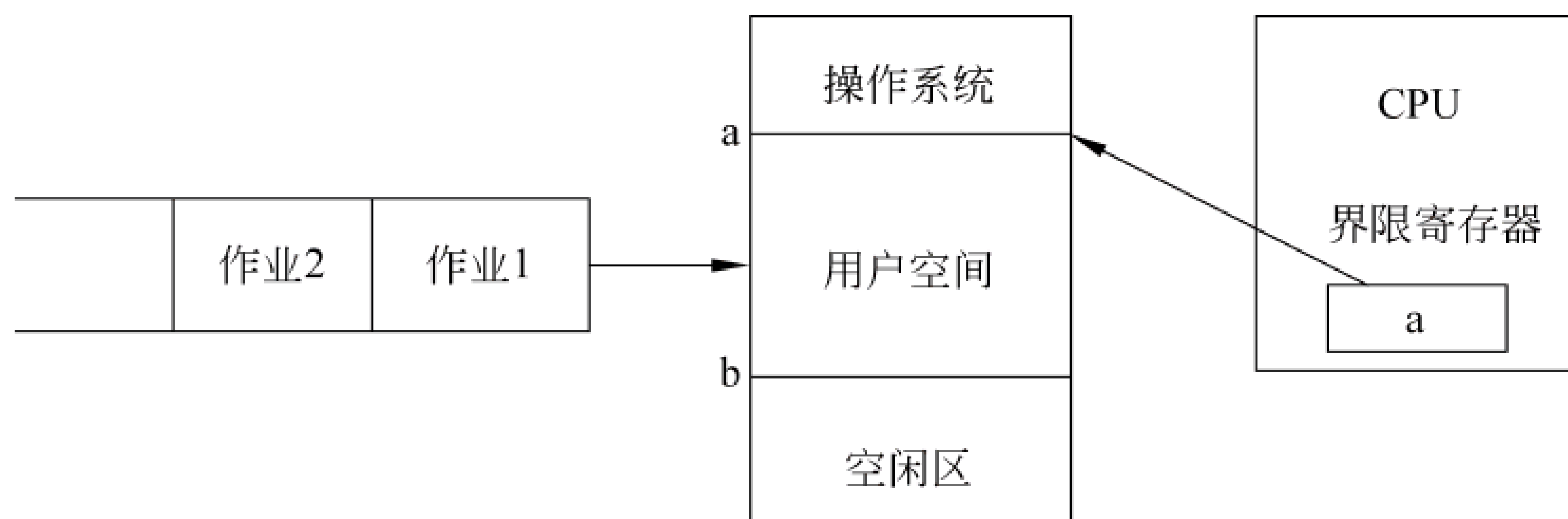


图 5.3 单分区存储管理示意图

采用这种方式管理时,处理器中设置一个界限寄存器,该寄存器的内容为用户区的起始地址。一般情况下,界限寄存器的内容是不变的,只当操作系统的功能扩充或修改时改变了所占区的大小,才更改界限寄存器的内容。

由于主存中只允许装入一个作业,所以那些等待装入主存的作业在外存中排成一个作业队列。当主存中无作业或一个作业执行结束,就可以让作业队列中的一个作业装入主存。

如果界限寄存器指示用户区的起始地址为 a,则作业总是被装到从 a 单元开始的一个主存连续区内。若作业的地址空间小于用户区,则作业占据用户区中的一部分,其余部分就成为空闲区。不管空闲区的大小如何都不再用来装另一个作业。若作业的地址空间大于用户区,可以采用覆盖技术控制作业的执行。

单分区的存储管理每次只允许一个作业装入主存,因此不必考虑作业在主存中的移动问题。因此可以采用静态重定位的方式进行地址转换,即在作业装入主存时,由装入程序完成地址转换,装入程序只要把界限寄存器的值加到逻辑地址上就可完成地址转换。

作业执行时,处理器要对每条指令中的绝对地址进行检查。若

$$界限地址 \leqslant 绝对地址 \leqslant 主存最大地址$$

则可执行,否则有地址错误,形成“地址越界”的程序性中断事件。这样就可以限定作业在规定主存区内执行,避免破坏操作系统的信息,达到存储保护的目的。

单分区的存储管理适用于单道程序的系统,这种管理方式有几个主要缺点:

(1) 当作业执行中出现了某个等待事件(例如等待外围设备传输数据)时,处理器就处于空闲状态,不能被利用。

(2) 一个作业独占主存中的用户区,当主存中有空闲区域时,也不能被其他作业利用,降低了主存空间的利用率。

(3) 外围设备也不能充分被利用。

若把单分区的存储管理方式用于分时系统中,则可采用交换技术让多个用户作业轮流进入主存储器执行。此时,多个用户的作业信息都保留在大容量的磁盘上,把其中的一个作

业先装入主存储器让它执行。当执行中出现等待事件或用完一个时间片,则把该作业从主存储器中“换出”,再把下一个轮到的作业“换入”到主存储器中执行。

值得注意的是,若对作业采用静态重定位方式完成地址转换,那么作业信息都由绝对地址来指示。所以,一旦作业被换出后,又再次被换入,则一定要把它装到与被换出前相同的主存空间位置,以保证按绝对地址正确执行。

5.3.2 多分区存储管理

多分区的存储管理是把主存中的用户区划分成若干个连续区域,每个连续区中可装入一个作业。因此,多分区存储管理适合多道程序系统。多分区存储管理可采用固定分区方式或可变分区方式进行管理。

1. 固定分区存储管理

固定分区管理方式是把主存中可分配的用户区域预先划分成若干个连续区,每个连续区的大小可以相同,也可以不同。但是,一旦划分好分区之后,主存中分区的个数和大小都将就固定下来,不能改变。

在固定分区方式管理下,每个分区可用来装入一个作业。由于主存中有多个分区,就可以在不同的分区中同时装入多个作业,这种管理方式适用于多道程序设计系统。

等待进入主存的作业排成队列,当主存中有空闲的分区时,依次从作业队列中选择一个能装入该分区的作业。当所有的分区都已装有作业,则其他的作业暂时不能再装入,绝不允许在同一分区中同时装入两个或两个以上的作业。已经被装入主存的作业得到处理器运行时,要限定它只能在所占的分区中执行。图5.4是划分成3个分区的固定分区管理方式的示意图。

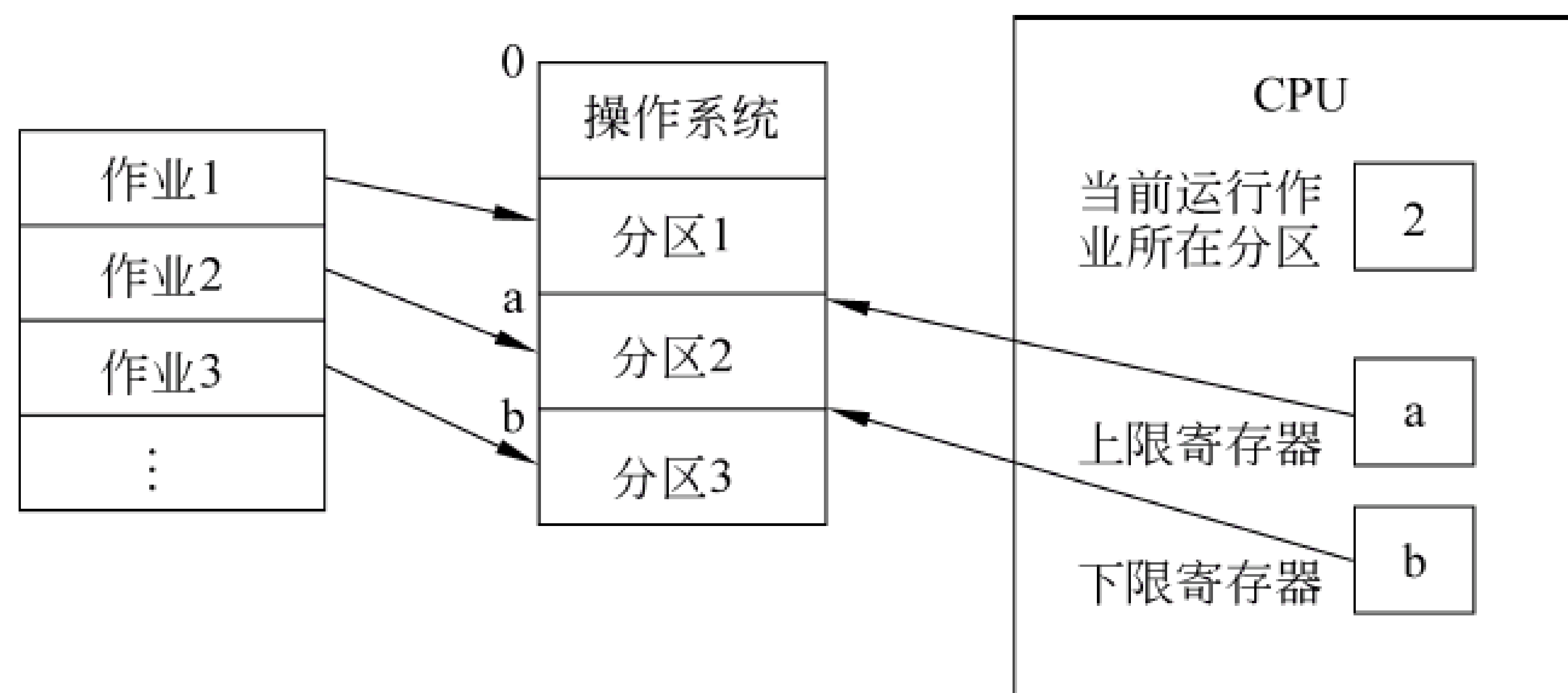


图5.4 固定分区存储管理示意图

1) 使用的数据结构

为了管理主存空间的使用,必须设置一张主存分配表,以记录各分区的使用情况。主存分配表中应指出各分区的起始地址和长度,并为每个分区设一个标志位。当标志位为“0”时,表示对应的分区是空闲分区,可以用来装作业;当标志位为非“0”时,表示对应的分区已被占用,为占用该分区的作业名。

主存分配表的一般格式如图5.5所示。

分区号	起始地址	长度	占用标志

图5.5 主存分配表的一般格式

2) 主存空间的分配与回收

当作业队列中有作业要装入主存时,存储管理可

采用顺序分配算法进行主存空间的分配。顺序查看主存分配表,找到一个标志为“0”的分区,再把欲装入作业的逻辑空间的大小与找到的分区长度进行比较。当找到的分区能容纳该作业时,则把此分区分配给该作业,把它的作业名填到占用标志位上;当找到的分区不能容纳该作业时,则重复上述过程继续顺序查看主存分配表中是否有能满足该作业长度要求的且标志为“0”的分区,若有,则分配;若无,则该作业暂时得不到主存空间而不能装入。固定分区分配算法如图 5.6 所示。

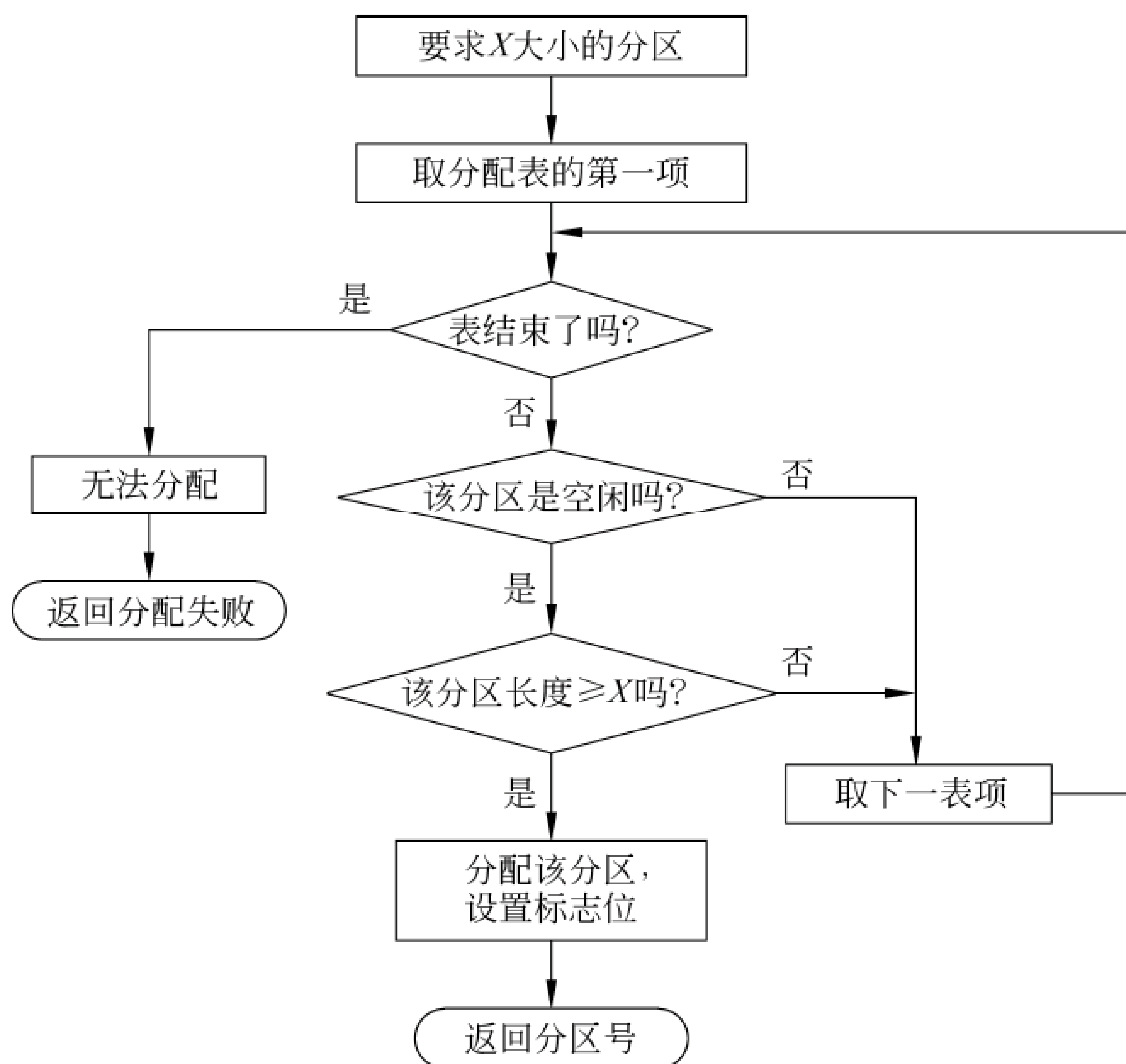


图 5.6 固定分区分配算法

装入分区的作业执行结束后必须归还所占用的分区,存储管理根据作业名查看主存分配表,从占用标志位中的记录可知道该作业所占用的分区,把该分区的占用标志位重新置成“0”,表示该分区现在又成了空闲区,可用来装入新作业。

3) 地址转换和存储保护

由于固定分区管理方式是预先把主存划分成若干个区,每个区又只能用来装入一个作业,因此,作业在执行过程中是不会被改变存放位置的。于是,可以采用静态重定位的方式把作业装入分配到的分区中去。由装入程序把作业中的逻辑地址与分区的起始地址(上限地址)相加,得到相应的绝对地址。装入程序在进行地址转换时要检查其绝对地址是否在指定的分区范围中,若是,则可把作业装入,否则不能装入该作业且要归还分配给该作业的分区。

一个装入主存的作业占用处理器时,进程调度程序必须把该作业所在分区的上限地址和下限地址存入处理器中的上限寄存器与下限寄存器中(如图 5.4 中所示)。处理器执行该作业时,对每条指令中的地址都要进行以下的核对:

上限地址≤绝对地址≤下限地址

如果绝对地址在上、下限地址范围内,则可按绝对地址访问主存;如果不在上、下限界地

址范围内，则形成“地址越界”的程序性中断事件，达到存储保护的目的。

一个作业让出处理器时，另一个作业可能被选中占用处理器。这时，应更改上、下限寄存器的内容，改为当前被选中作业所在分区的上限地址和下限地址，以保证处理器能控制作业在规定的分区内执行。

4）内存扩充

可以采用覆盖或交换技术来扩充内存。

5）主存空间的利用率

用固定分区方式管理主存时，总是为作业分配一个不小于作业长度的分区。因此，实际上很多作业只占用了分区的一部分空间，使分区中有一部分空间闲置不用，被称为内部碎片，影响主存空间的利用率。

采用如下几种办法可使主存空间利用率得到改善：

（1）划分分区时按分区的大小顺序排列，低地址部分是较小的分区，高地址部分是较大的分区。各分区按从小到大的次序依次登录在主存分配表中。这样，在采用顺序分配算法时，从当前的空闲区中总是找出一个能满足作业要求的最小空闲区分配给作业。一方面使闲置的空间尽可能地减少，另一方面尽可能地保留较大的空闲区，以便有大作业请求装入时容易得到满足。

（2）根据经常出现的作业的大小和频率划分分区，这样能使主存空间的利用率提高。但这种方法实现比较困难。

（3）按作业对主存空间的需求量排成多个作业队列，规定每个作业队列中的各作业只能依次装入对应的指定分区中；不同的分区中可同时装入作业；某作业队列为空时，该作业队列对应的分区也不能用来装其他作业队列中的作业。采用这种分区法有效地防止了小作业进入大分区，从而减少了闲置的主存空间。但是，如果分区划分不合适，则会造成某个作业队列经常是空队列，那么，对应的分区经常没有作业被装入，反而使分区的利用率不高。所以，采用多个作业队列的固定分区法时，可结合作业的大小和出现的频率划分分区，以达到期望的利用率。

2. 可变分区存储管理

1）基本原理与使用的数据结构

可变分区存储管理方式不是预先把主存储器进行分区，而是在作业要求装入时，根据作业需要的主存量和当时主存空间的使用情况决定是否可以装入该作业。当主存中有足够的空间能满足作业需求时，则按作业需求量划出一个分区分配给该作业。由于分区的大小是按作业的实际需求量来定的，故分区的长度不是预先固定的，且分区的个数也是变化的。

可变分区管理方式使用的数据结构主要有已分配区表和未分配表。

已分配区表记录已装入的作业在主存中占用分区的始址和长度，用标志位指出占用分区的作业名或本条目为空。基本格式如图 5.7(a)所示。

未分配表记录主存中可供分配的空闲区。可以用空闲区表或空闲块链实现。空闲区表的形式如图 5.7(b)所示，包含空闲块的始址和长度，也用标志位指出该分区是未分配的空闲区或本条目为空。由于已占分区和空闲区的个数不定，因此，两张表格中都应设置适当的空栏目（置标志位的状态为空），分别用以登记待装入主存的作业占用的分区和作业撤离后的新空闲区。

始址	长度	标志位
…	…	作业名/空

(a) 已分配区表

始址	长度	标志位
…	…	未分配/空

(b) 空闲区表

图 5.7 可变分区管理方式的主存分配表

空闲块链是由链表实现的,链表中的每一节点记录一个空闲块的大小和下一空闲块的始址,链表的头指针指示第一个空闲块的始址。形式如图 5.8 所示,图中最末一个节点中的“^”表示此空闲区是最后一个空闲区。

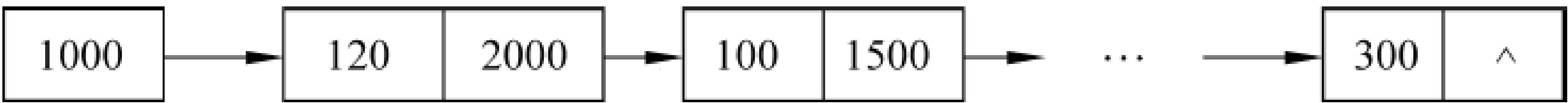


图 5.8 空闲块链示例

下面以空闲区表记录未分配的空闲区来讨论各问题。

2) 主存空间的分配与回收

系统初始化时,把整个用户区看作是一个大的空闲区。这时已分配区表中的所有条目的标志位都为“空”。空闲区表中第一条目的始址为用户区的开始地址,长度为用户区的总长度,标志位为“未分配”;其余条目的标志位为“空”。

当要装入一个作业时,从空闲区表中查找标志为“未分配”的空闲区,从中找出一个能容纳该作业的空闲区。如果找到的空闲区正好等于该作业的长度,则把该区全部分配给作业。这时应把该空闲区登记栏中的标志改为“空”状态,同时在已分配区表中找一个标志为“空”的栏目登记新装入的作业占用分区的始址、长度和作业名。如果找到的空闲区大于作业长度,则把空闲区分成两部分,一部分用来装作业,另一部分仍为空闲区,这时应修改已分配区表和空闲区表中相应的内容。

可变分区方式常用的主存分配算法有“最先适应”、“最优适应”和“最坏适应”。

(1) 最先适应分配算法

最先适应分配算法在每次分配时,总是顺序查找空闲区表,找到第一个能满足作业长度要求的空闲区,如果空闲区大小与作业大小相同,则直接分配给该作业,同时修改已分配区表和空闲区表;如果空闲区大小比作业长度大,分割这个找到的空闲区,一部分分配给作业,另一部分仍为空闲区,同时修改已分配区表和空闲区表。

采用最先适应分配算法,可把空闲区按始址顺序从低到高登记在空闲区表中。于是,在分配时总是尽量利用低地址部分的空闲区,而使高地址部分保持有较大的空闲区,有利于大作业的装入。但是,这时每当有作业归还分区时,必须调整空闲区表,把归还区按地址顺序插入到空闲区表的适当位置。

(2) 最优适应分配算法

最优适应分配算法总是按作业要求挑选一个能满足作业要求的最小空闲区,这样保证可以不去分割一个更大的区域,使装入大作业时比较容易得到满足。在实现这种算法时,可把空闲区按长度以递增顺序登记在空闲区表中。这样,每当收回一个分区.必须把收回后的分区按长度顺序插入到空闲区表的适当位置。实现最优适应分配算法时,可把空闲区按大小顺序从小到大登记在空闲区表中。最优适应分配算法如图 5.9 所示。

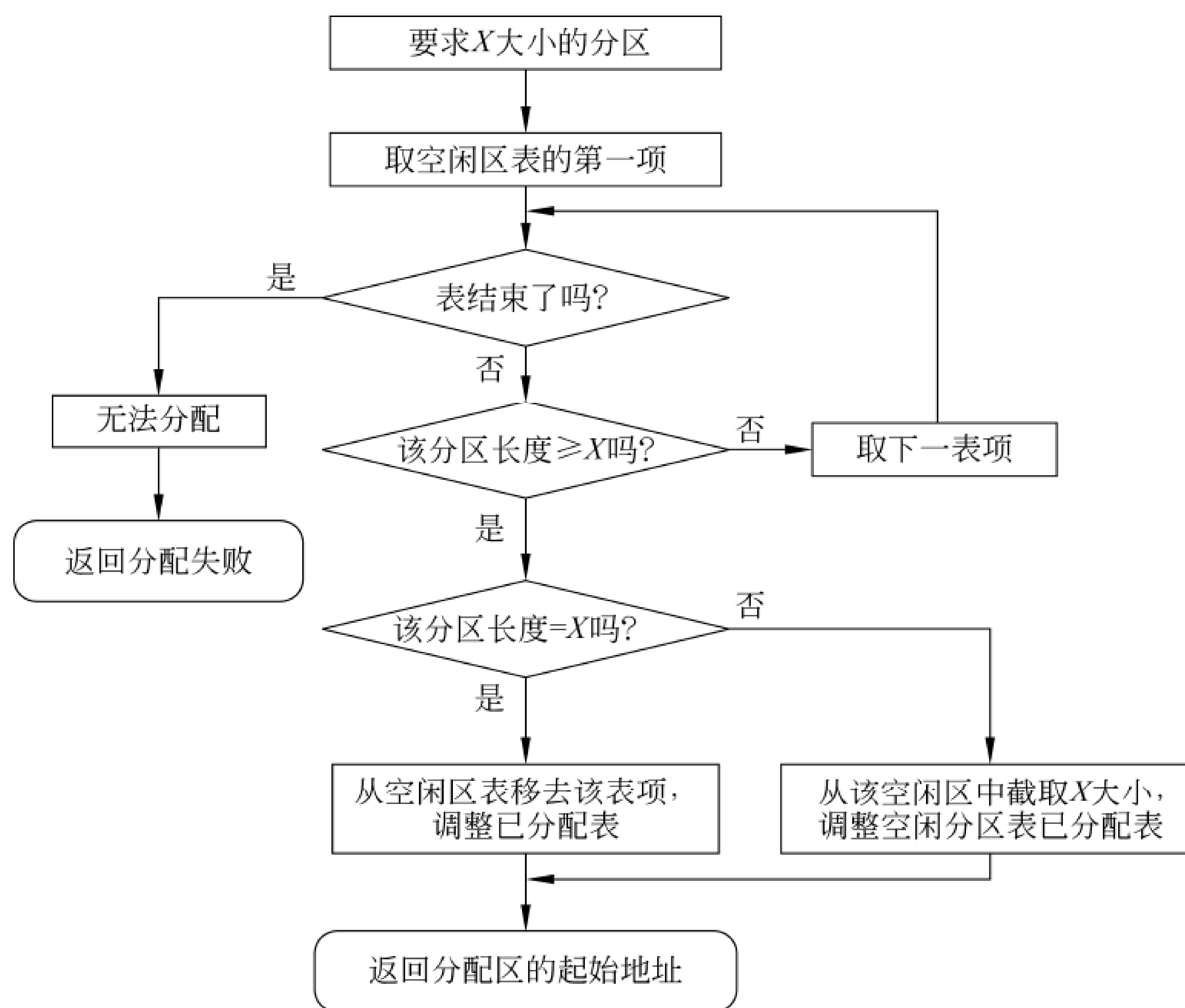


图 5.9　最优适应算法

采用最优适应分配算法，有时找到的一个分区可能只比作业所要求的长度略大一些。这样经分割后剩下的空闲区（即“碎片”）就很小了，这种碎片往往是无法使用的，影响了主存空间的使用率。

（3）最坏适应分配算法

最坏适应分配算法总是挑选一个最大的空闲区分割一部分给作业使用，使剩余部分的空闲空间不至于太小，仍可供分配使用。这种分配算法对中小型作业是有利的。实现最坏适应分配算法时，空闲区表中的登记项可按空闲区长度以递减顺序排列。

当有作业执行结束撤离时，同样要修改两张表格。从已分配区表中找出作业占用的分区，把表中条目的状态改成“空”，而将归还的分区登记到空闲区表中。

当一个作业运行结束，归还所占分区。一个归还区可能有上邻空闲区（始址小于归还区的始址），也可以有下邻空闲区（始址大于归还区的始址），或既有上邻又有下邻空闲区，或既无上邻空闲区又无下邻空闲区。这样就出现了空闲区的合并问题，合并情况如图 5.10 所示。

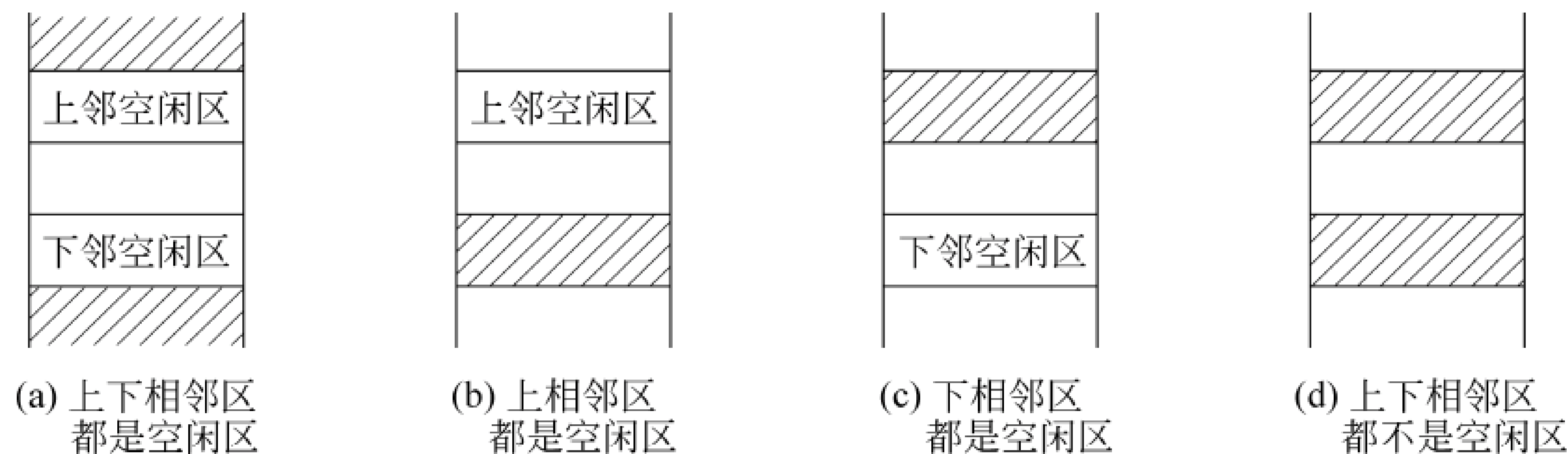


图 5.10　空闲区的合并

在回收时应顺序检查空闲区表中标志为“未分配”的栏目，以确定是否有相邻空闲区。假定作业归还的分区始址为 S，长度为 L，则：

(1) 归还区有下邻空闲区。如果 $S+L$ 正好等于空闲区表中某个登记栏目（假定为第 i 栏）所示分区的始址，则表明归还区有一个下邻空闲区，应进行合并。这时只要修改第 i 栏登记项的内容：始址＝S，长度＝原长度＋L 即可。

(2) 归还区有上邻空闲区。如果空闲区表中第 i 个登记栏中的“始址＋长度”正好等于 S，则表明归还区有一个上邻空闲区，应进行合并。这时也要修改第 i 栏登记项的内容：始址不变，长度＝原长度＋L 即可。

(3) 归还区既有上邻空闲区又有下邻空闲区。如果 S＝第 i 栏始址＋长度并且 $S+L$＝第 k 栏始址，表明归还区既有上邻空闲区，又有下邻空闲区，应把 3 个区合并成一个空闲区登记入空闲区表中。进行如下操作：第 i 栏始址不变，长度＝i 栏中原长度＋k 栏中长度＋L；第 k 栏的标志位应修改成“空”状态。

(4) 归还区既无上邻空闲区又无下邻空闲区。如果在检查空闲区表时，无上述 3 种情况出现，则表明归还区既无上邻空闲区又无下邻空闲区。这时，应找一个标志位为“空”的登记栏；把归还区的始址和长度登记入表，且把该栏目中的标志位修改成“未分配”。

图 5.11 给出了空闲区合并算法流程图，该算法中空闲区表项按空闲区的开始地址由小到大排序。

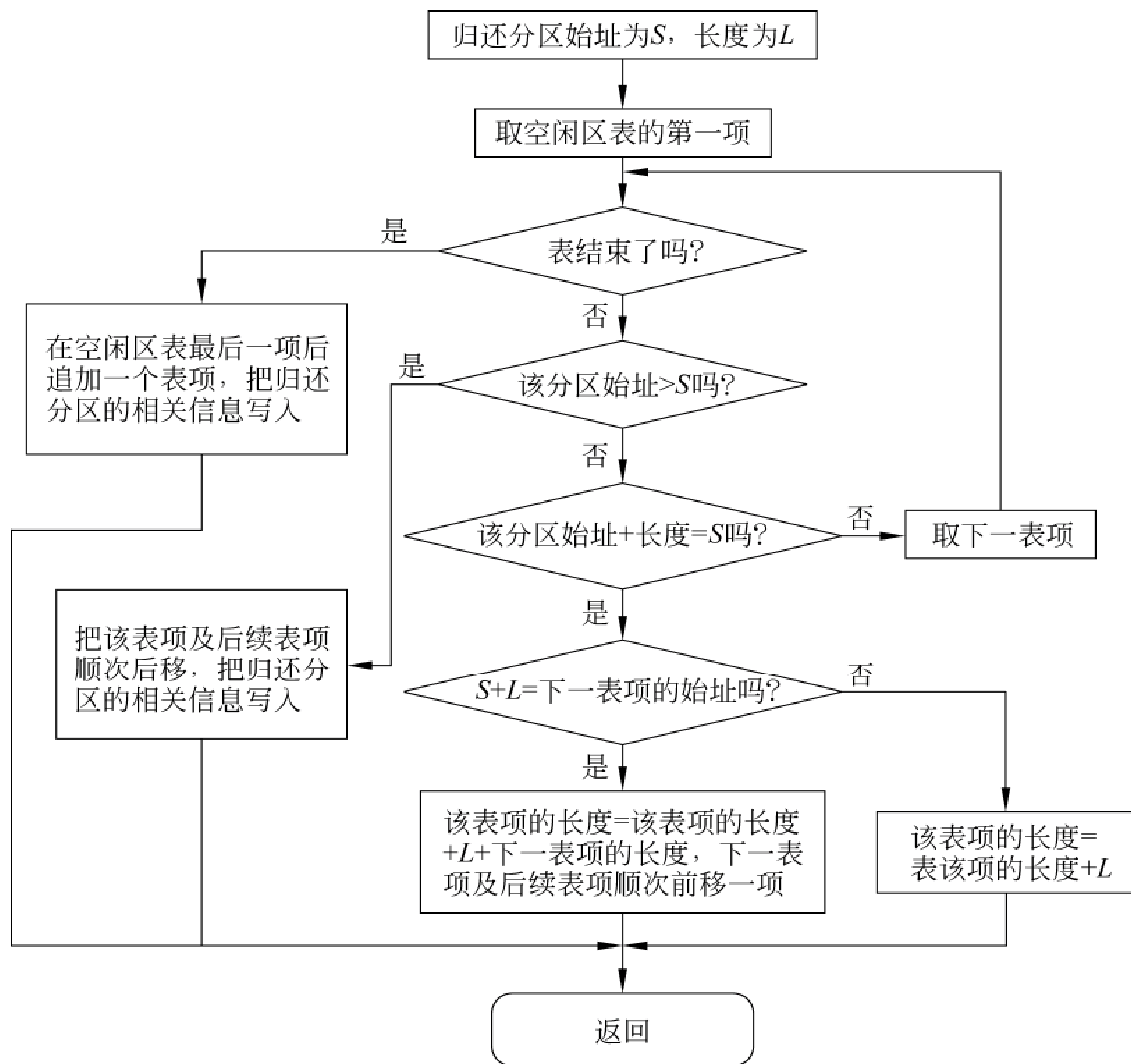


图 5.11　空闲区合并算法

3）地址转换和存储保护

采用可变分区方式管理时,一般均采用动态重定位方式装入作业。因此,要有硬件的地址转换机构作支持。硬件设置两个专用的控制寄存器:基址寄存器和限长寄存器。限长寄存器用来存放作业所占分区的长度,基址寄存器用来存放作业所占分区的起始地址。

当作业被装到所分配的分区后,把分区的起始地址和长度作为现场信息存入该作业进程的 PCB 中。作业调度选中某作业占用处理器时,作为现场信息的分区始址和长度被送入基址寄存器和限长寄存器中。作业执行过程中,处理器每执行一条指令都要取出该指令中的逻辑地址,当逻辑地址小于或等于限长寄存器的值时,则逻辑地址加基址寄存器值就可得到绝对地址;当逻辑地址大于限长寄存器的值时,表示欲访问的主存地址超出了所分配的分区范围。这时就不允许访问,形成一个“地址越界”的程序性中断事件,达到存储保护的目的。图 5.12 是地址转换的示例图。

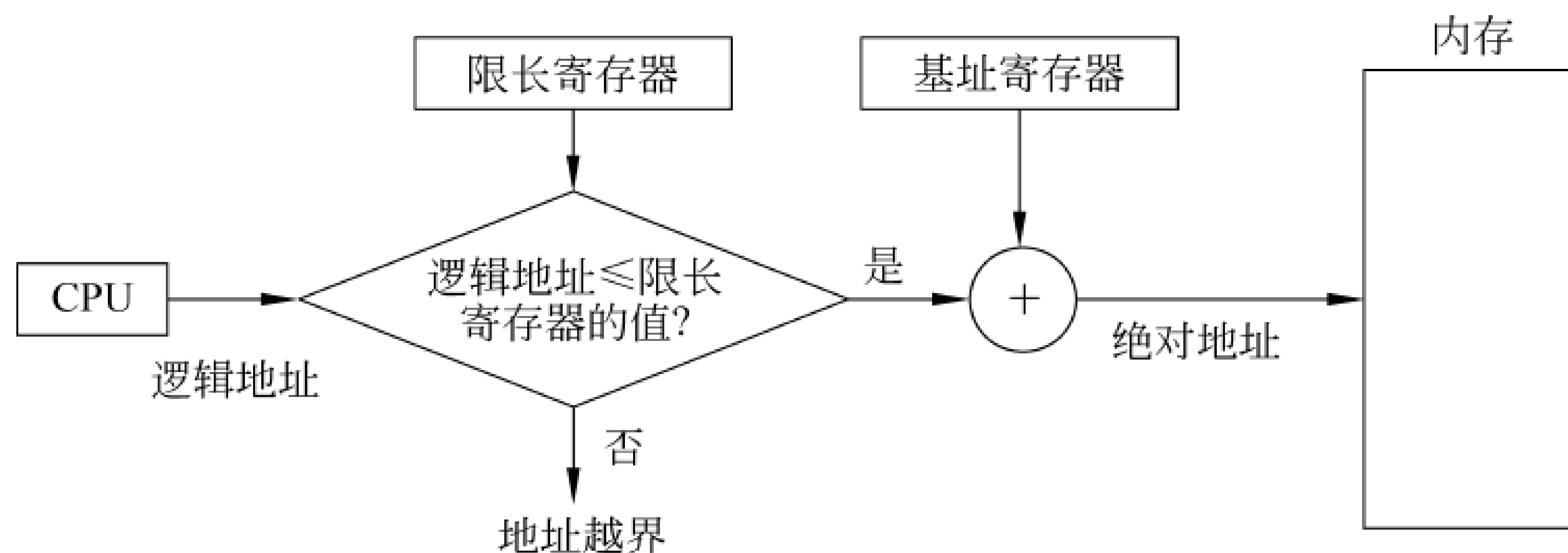


图 5.12　可变分区存储管理的地址转换

另外也可以用保护键法对内存各区提供保护。这样已分配区表可以改进为如图 5.13 所示的形式。

始址	长度	标志	保护位

图 5.13　已分配区表

4）内存扩充和移动技术

在分区管理方式中,可以采用覆盖或交换技术来对内存进行扩充,另外也可采用移动技术使分散的空闲区集中起来,以容纳新的作业,也为作业执行过程中扩充主存空间提供方便。当一道作业在执行中要求增加主存时,只要适当移动邻近的作业就可增加它所占的分区长度。

移动可以集中分散的空闲区,提高主存空间的利用率。但是,采用移动技术时必须注意下列问题:

(1) 移动会增加系统开销,所以应尽量减少移动。

(2) 移动是有条件的。不是任意一个作业都能随意移动。例如,某个作业在执行过程中正在等待外围设备传输信息,那么就不能移动该作业。这是因为外围设备与主存储器之间的信息交换是按确定了的主存绝对地址进行传输的,如果这时改变了作业的存放区域,则作业就得不到从外围设备传送来的信息,或不能把正确的信息传送到外围设备。所以,移动一道作业时,应先判定它是否在与外围设备交换信息。若为否,则可以移动该作业;若是,则暂不能移动该作业,必须待信息交换结束后才可移动。

采用移动技术时应尽可能地减少移动的作业数和信息量。

一种可行的办法是改变作业装入主存的方式,因为作业的装入方式对移动的作业数和

信息量是有影响的。图 5.14 给出 4 道作业被装入主存时的存放方式。

采用图 5.14(b)的装入方式比采用图 5.14(a)的装入方式减少移动量。比如,当作业 J1 执行结束后,它所占的分区为空闲区。现在主存中有两个空闲区。现有一个新的作业 J5 要求装入,若 J5 要求的主存量比任何一个空闲区大,但不超过两个空闲区的总量,这时可移动已在主存中的作业,集中空闲区,按图 5.14(a)的方式必须移动 3 道作业,而按图 5.14(b)的方式只要移动一道作业。

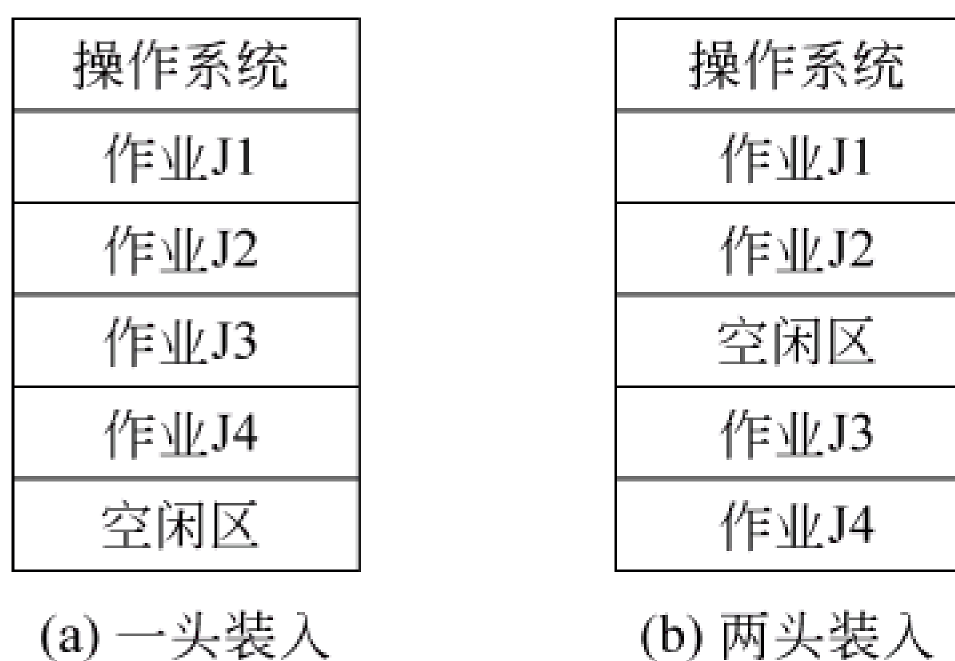


图 5.14 作业的装入方式

5) 内存信息共享问题

在多道程序系统中,有许多程序和数据是可以共享的。例如,编译程序、编辑程序、公共子程序和公用数据等都是可共享的。这些共享的信息在主存中只需要保留一个副本。

程序共享可节省主存空间,但对每个作业来说,在执行时,应既可访问自己所在的分区又可访问公共区域。在这种情况下,硬件应该提供两组限长寄存器和基址寄存器对,其中一组用来存放共享区的长度和始址,另一组用来存放占用处理器的那个作业所在分区的长度和始址。

一般说,共享信息只能读或调用执行,不允许修改。因此,处理器执行的指令若是访问共享区,需核对该指令的操作要求。如果是"写"操作,则立即停止指令的执行,且形成一个"非法操作"的程序性中断事件,以达到存储保护的目的。

5.3.3 分区存储管理的评价

分区存储管理的主要优点如下:

(1) 实现了多个作业或进程对内存的共享,有助于多道程序设计,从而提高了系统资源的利用率。

(2) 该方法要求的硬件支持少,管理算法简单,因而实现容易。

其主要缺点如下:

(1) 内存利用率仍然不高。如果不采用移动技术,就存在着严重的碎片不能利用的问题,影响了内存的利用率。

(2) 作业或进程的大小受分区大小控制,除非配合采用覆盖和交换技术。

(3) 虽能实现信息共享,但比较困难。

5.4 页式存储管理

用分区方式管理主存时,每道作业都占用主存的一个连续的存储空间。因此,当主存中无足够大的连续空间时,作业或进程就无法装入,必须移动已在主存中的某些作业或进程后才能再装入新的作业或进程,这样不仅不方便,而且系统开销大。是否可以把逻辑地址连续的作业或进程分散存放到几个不连续的主存区域中,但仍能保证作业或进程的正确执行?解决这一问题的一种行之有效的存储管理方式是页式存储管理。

页式存储管理方法分为静态页式存储管理和动态页式存储管理。

5.4.1 页式存储管理的基本原理

1. 基本原理

页式存储管理需要硬件的支持，首先把主存分成大小相等的许多区，把每个区称为块，块是进行主存空间分配的物理单位。程序的逻辑空间进行分页，页的大小与块的大小一致。这样，就可把作业或进程按页存放到块中。页式存储管理提供给编程使用的逻辑地址由两部分组成：页号和页内地址，其格式如图 5.15 所示。

页号	页内地址

图 5.15 页式存储管理提供编程使用的逻辑地址

地址结构确定了主存分块的大小（由页内地址所占的位数确定），也就决定了页的大小。同时确定了一个作业或进程所占的最多页数（由页号地址所占的位数确定）。假定地址用 m 个二进制位表示，其中页内地址部分占用了 $n(n<m)$ 个二进制位。那么，每一个块的长度就是 2^n，页号部分占用了 $m-n$ 位，所以，最大的作业或进程可允许有 2^{m-n} 个页面。从地址结构来看，逻辑地址仍是连续的。逻辑地址从“0”开始（页号为“0”，页内地址也为“0”），当编址到 2^n-1 时，第 0 页的页内地址的各位均为“1”，即占满了一个页面。下一个地址是 2^n，这时页号部分就为“1”，而页内地址部分又恢复到“0”，表示进入了第 1 页。依次类推，一组顺序的逻辑地址由地址结构将其自然地分页了，所以，用户编程时无须考虑如何分页的问题，仍使用连续的逻辑地址即可。页式存储管理的逻辑地址是一维的。

2. 使用的数据结构

在页式存储管理方法中使用了 3 种主要数据结构：页表、请求表和存储块表。

1）页表

简单的页表由页号和块号组成，用来记录一个作业或进程占用块的情况。页表中的表项数目与进程或作业所拥有的页的数目相等。每个作业或进程有一张页表。

根据需要在页表中可以增设项目，如权限标志位、内外存标志位、改变位和外存始址等。

2）请求表

请求表用来记录作业或进程申请或占用内存情况。它记录每个作业或进程页表起始地址和长度，以进行内存分配和地址变换。另外，请求表中还应包括每个作业或进程所要求的块数。整个系统使用一张请求表，如图 5.16 所示。

进程号	请求块数	页表始址	页表长度	状态
…	…	…	…	已分配/未分配

图 5.16 请求表示例

3）存储块表

存储块表指出内存各块的状态，整个系统使用一张表。存储块表主要有 3 种实现方法：位示图法、空闲块表法和空闲块链法。

位示图法是在内存中划分一个固定区域，每个单元的每个比特位表示一个块的状态。如果某块已分配，则对应比特位置“1”，否则置“0”。位示图的形式如图 5.17 所示。

31	30	29	…	3	2	1	0	
1	0	0	…	1	0	1	1	0
0	1	1	…	0	0	1	0	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
空闲块总数：1290								

图5.17　位示图示例

空闲块表法是通过一张表记录内存中的空闲块，表中的每一项记录一个连续空闲块的首块号和连续空闲块的块数。

空闲块链法是通过一个链表记录内存中的空闲块，链表中的每一节点包含一个连续空闲块区域中空闲块总数和下一空闲块区域的首块位置，链表头指针记录第一个空闲区域的首块位置。空闲块的形式如图5.18所示，图中最末一个节点中的“0”表示再没有空闲的页。

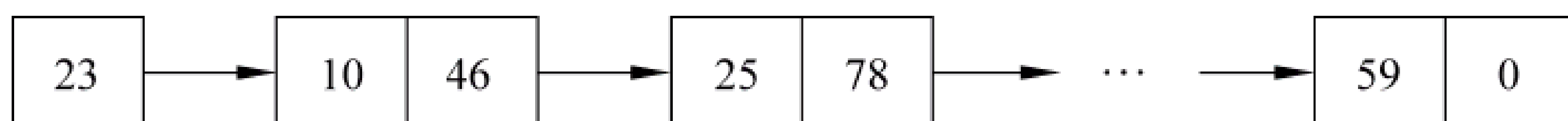


图5.18　空闲块链表示例

5.4.2　静态页式存储管理

静态页式存储管理是在作业或进程开始执行之前，把该作业或进程的程序段或数据段按页全部装入内存，并通过数据结构和地址转换机构来实现相应的管理。

1. 存储空间的分配与回收

这里以位示图方式的存储块表为例说明存储空间的分配与回收。

当一个作业或进程被选中，首先通过位示图查看内存中的空闲块是否满足该作业或进程的需求，如果不满足，则该进程或作业等待；若能满足，则根据需求从位示图中找出为“0”的位，个数与所需的块数相等，且把这些位置成“1”，从空闲块总数中减去本次分配的块数，按找到的位计算出对应的块号(块号＝字号×字长＋位号)，把作业或进程装入到这些块中，并为作业或进程建立一张“页表”。

当一个作业或进程执行结束，则应收回作业或进程所占的主存块。通过页表查到该作业或进程所占的块号，归还这些块。根据归还的块号计算出该块在位示图中对应的位置，假定归还块的块号为i，则在位示图中对应的位置为：字号＝[i/字长]，位号＝i mod 字长，将占用标志修改成“0”，把归还块数加入到空闲块总数中。

2. 地址转换

静态页式存储管理采用动态重定位方式装入作业或进程，因而要有硬件地址转换机构的支持，页表是硬件进行地址转换的依据。每执行一条指令时按逻辑地址中的页号查页表，若页表中无此页号，则产生一个“地址错”的程序性中断事件；若页表中有此页号，则可得到对应的主存块号，按计算公式(绝对地址＝块号×块长＋页内地址)可转换成欲访问的主存绝对地址。地址转换关系如图5.19所示。

在多道程序设计的系统中，进入主存储器的每个作业或进程都有一张页表。当某个作

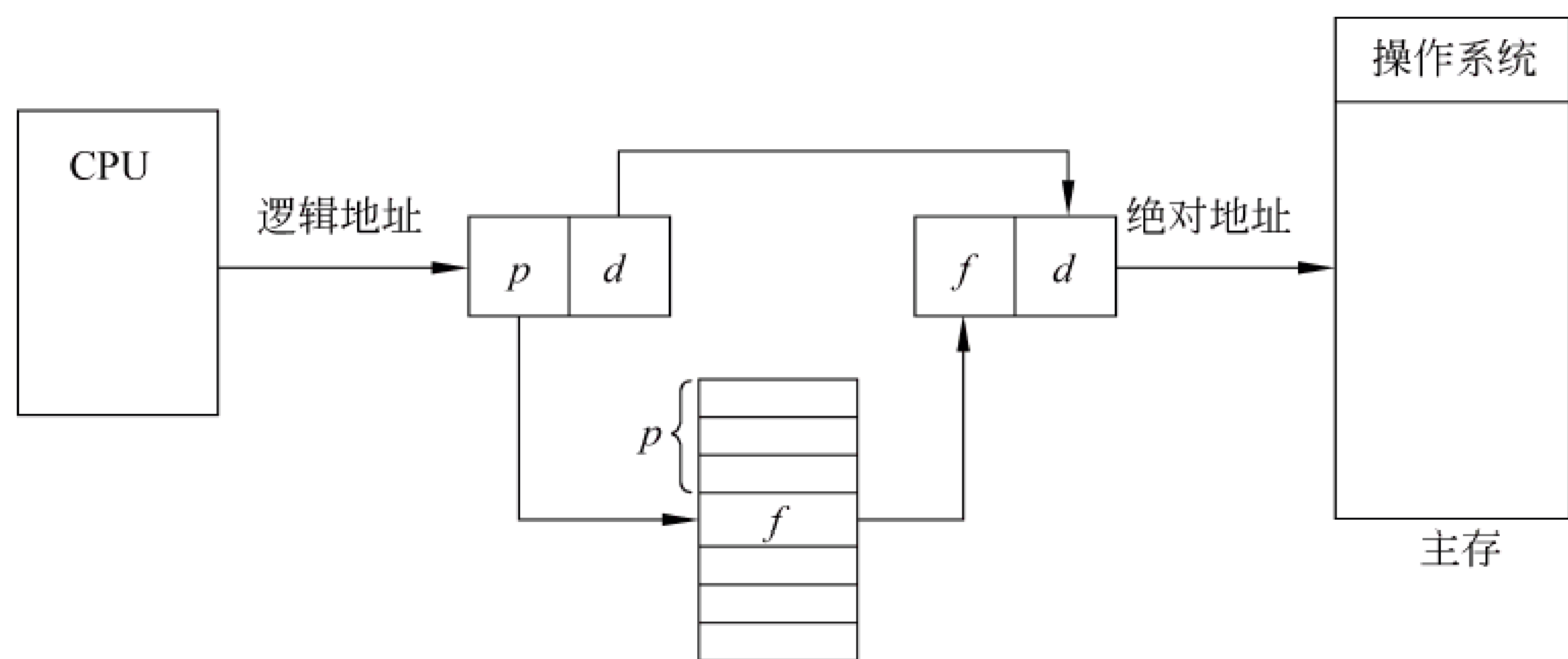


图 5.19 静态页式管理地址转换

业或进程可占用处理器时,则必须按该作业或进程的页表进行地址转换,以保证每个作业或进程的正确执行。为此,硬件要增加一个"页表控制寄存器",页表的始址存入基址寄存器中,页表的长度存入限长寄存器中。调度程序选中某个作业或进程占用处理器时,通过请求表查找该作业或进程的页表起始地址和长度,送入页表控制寄存器。地址转换机构根据页表控制寄存器就可找到该作业或进程的页表。

3. 快表

页式存储管理中的页表一般是存放在主存中的,于是,当要按给定的逻辑地址进行读写时,必须访问两次主存。第一次按页号读出页表中对应的块号,第二次按计算出来的绝对地址进行读写,这样就延长了指令的执行周期,降低了执行速度。

为了提高存取速度,通常把页表中的一部分内容存放在高速缓冲存储器(Cache)中,这部分页表称"快表"。高速缓冲存储器的造价很高,故一般容量较小。用于快表的高速缓冲存储器一般为8～16个存储单元;当含有8个单元时,平均命中率为85%;当含有12个单元时,平均命中率为90%;当含有16个单元时,平均命中率为97%。由此可见使用快表的实际效果还是很令人满意的。

有了快表之后,根据逻辑地址中的页号同时查找快表和页表,若该页已登记在快表中,按快表中的对应块号与逻辑地址中的页内地址形成绝对地址;若该页没有登记在快表中,则从页表中得到对应的块号,再与逻辑地址中的页内地址合成绝对地址,同时将该页登记到快表中。由于快表容量较小,当快表被填满后,又要在快表中登记新页时,需要淘汰快表中的一个旧登记项,最简单的淘汰策略是"先进先出",即总是淘汰最先登记的那一页。

整个系统只设置一个高速缓冲存储器,只有占用处理器者才能使用它。由于快表是动态变化的,所以,当占用处理器的作业或进程让出处理器时,应把该作业或进程的快表保存到它的进程控制块中。当再次占用处理器时,就可把它的快表恢复到高速缓冲存储器中。

4. 页的共享和保护

采用页式存储管理能方便地实现程序和数据的共享,被共享的信息存放在主存的若干块中,对作业或进程中涉及共享信息的页,使其页表中的有关表目指向共享信息的主存块,则多个作业或进程就可共享主存块中的信息。由于页式存储管理中的页不是按逻辑程序段进行划分的,所以对逻辑程序段的共享还是困难的,即只能按页共享。

页的共享可节省主存空间,但实现信息共享必须解决共享信息的保护问题。对共享的

数据不能修改。通常的办法是在页表中增加一些标志位，指出对该页信息可执行的操作。当违反规定的访问权限时，将产生一个“非法操作”的程序性中断事件。

5.4.3 动态页式存储管理

1. 动态页式存储管理

动态页式存储管理是把作业或进程信息作为副本存放在磁盘上，作业或进程执行时，把作业或进程信息的部分页面装入主存。作业或进程执行时，若所访问的页面已在主存中，则按静态页式存储管理方式进行地址转换，得到欲访问的主存绝对地址；若欲访问的页面不在主存中，则产生一个“缺页中断”(关于中断，请参阅第 7 章)，由操作系统把当前所需的页面装入主存中。

为此，在装入作业或进程时，就应在该作业或进程的页表中指出哪些页已在主存中，哪些页还没有装入主存。页表的格式如图 5.20 所示。

在作业或进程执行中访问某页时，硬件的地址转换机构查页表，若该页对应标志位为“1”(表示在主存)，则按该页对应的主存块号进行地址转换，得到绝对地址；若该页对应标志位为“0”(表示不在主存)，则硬件形成“缺页中断”。当中断装置响应该中断后，操作系统就处理这个缺页中断。

缺页处理过程简单阐述如下：

(1) 根据当前执行指令中的逻辑地址查页表，判断该页是否在主存储器中。

(2) 该页标志位为“0”，产生缺页中断。

(3) 操作系统响应并处理该缺页中断；处理的办法是查存储表，找一个空闲的主存块，查页表找出该页在磁盘上的位置，启动磁盘读出该页信息。

(4) 把从磁盘上读出的信息装入找到的空闲主存块中。

(5) 修改页表中对应的表目。填上该页所占用的主存块号，把标志位置为“1”，表示该页已在主存中。

(6) 由于产生缺页中断时的那条指令并没执行完，所以在把页面装入之后应重新执行被中断的指令。当重新执行该指令时，由于要访问的页面已被装入主存，所以可正常执行下去。

在上述过程中，假定分配给作业或进程的主存块中有空闲的。如果无空闲块，则可选择一个已在主存中的页，把它暂时调出主存。若在执行中该页面被修改过，则把该页信息重新写回到磁盘上，否则不必重新写回磁盘。当一页被暂时调出主存后，让出的主存空间用来存放当前需要使用的页面。为此页表可改进为如图 5.21 所示。

页号	页面号	标志位	外存地址

图 5.20　页表的格式

页号	页面号	标志位	外存地址	改变位

图 5.21　改进的页表格式

上面讨论的在作业或进程执行过程中某条指令不在内存而发出中断请求，从而进行页的调入的方式，通常称为“请求调入”；还有一种称为“预调入”的方式。

预调入方式是系统对那些在外存中的页进行调入顺序计算，估计出这些页中指令和数据的执行和被访问的顺序，并按此顺序将它们调入内存。

预调入方式与请求调入方式除了在调入方式上有些差异外，其他方面基本相同，在此不

做详细介绍。

在动态页式存储管理中,为了加快地址的转换速度,也可把页表中的一部分放入高速缓存中,形成快表,其操作方式与静态页式存储管理中的快表相同。

动态页式管理的流程图如图5.22所示,图中有关地址变换部分是由硬件自动完成的。当硬件变换机构发现所要求的页不在内存时,产生缺页中断信号,由中断处理程序做出相应的处理。中断处理程序是由软件实现的。除了在没有空闲页面时要按照置换算法选择出被淘汰页面之外,还要从外存读入所需要的虚页。这个过程要启动相应的外存并涉及文件系统。因此,请求页式管理是一个十分复杂的处理过程,内存利用率的提高是以牺牲系统开销的代价换来的。

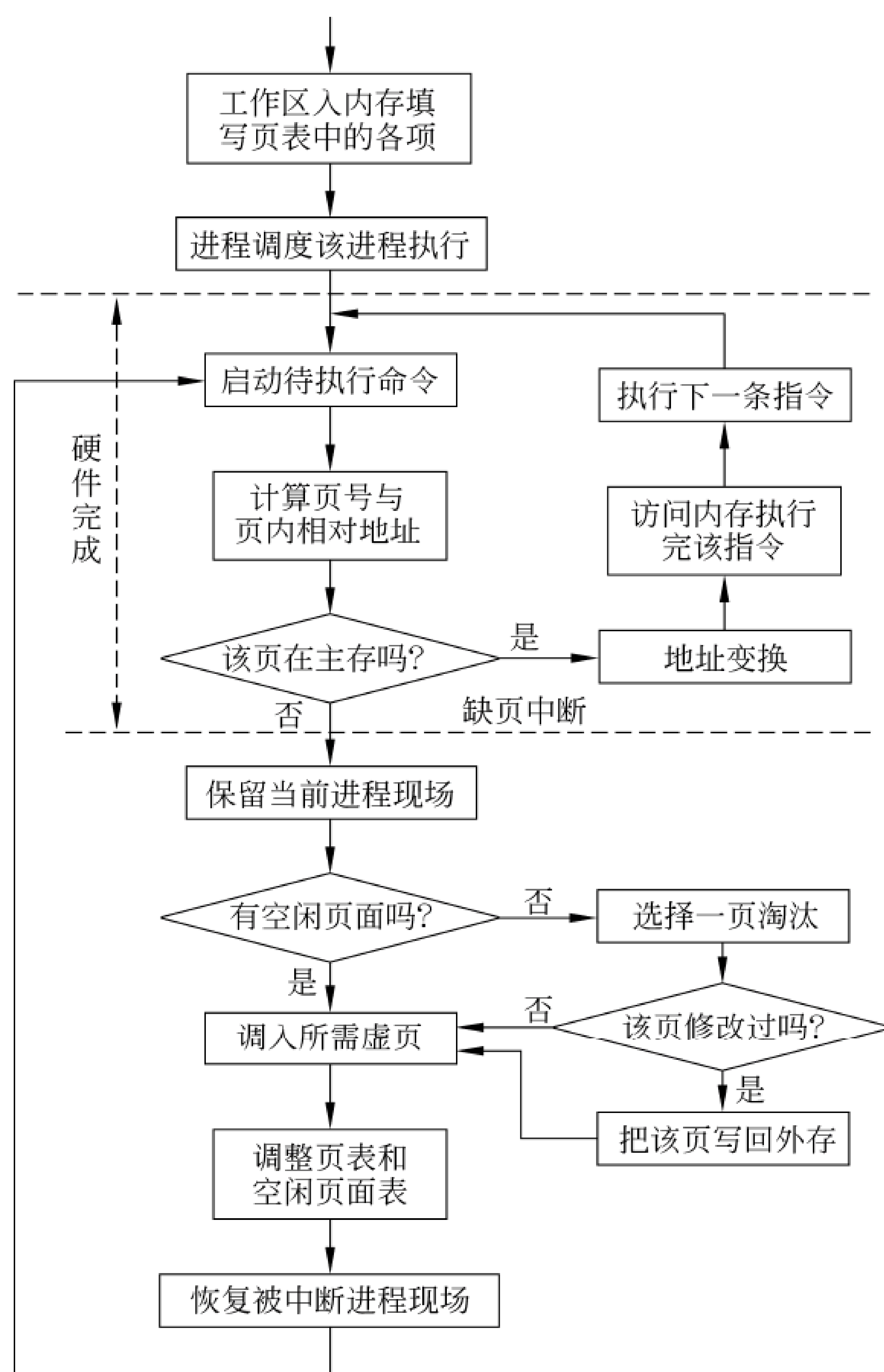


图5.22　动态页式存储管理的流程图

2. 页面置换

当主存中无空闲块时,为了装入一个页面而必须按某种算法从已在主存的页中选择一页,将它暂时调出主存,让出主存空间,用来存放所需装入的页面,这个工作称为页面置换。常用的页面置换算法有随机置换算法、轮转置换算法、先进先出置换算法、最近最少用置换算法、最近最不常用置换算法、Clock置换算法和最佳置换算法。

1）随机置换算法(random algorithm)

在系统设计人员认为无法确定主存中哪些页被访问的概率低时，随机地选择某个用户的页面并将其换出是一种明智的做法。

2）轮转置换算法(Round Robin，RR)

该算法循环换出用户区内的一个可以被换出的页，无论该页是刚被换进或已换进很长时间。该算法内存的利用率不高。

3）先进先出置换算法(First-In First-Out，FIFO)

该算法总是选择最先装入主存储器的那一页调出，或者说是把驻留在主存中时间最长的那一页调出。

4）最近最少用置换算法(Least Recently Used，LRU)

该算法是基于程序执行的局部性，即程序一旦访问到某些数据和指令时可能在一段时间里还经常会访问它们，因此，不应该把这些页面调出。如果某个内存页在最近一段时间里没有被访问过，则在最近的将来也可能暂时不会被访问，所以，需要装入新页时可以选择这些页面调出。

5）最近最不常用置换算法(Least Frequently Used，LFU)

该算法是根据在一段时间里页面被使用的次数选择可以调出的页。如果一个页面被访问的次数多，则是经常要使用的页面，就不应该把它调出。所以，这种算法总是选择被访问次数少的页面调出。

一种简单的实现方法是为每一页设置一个计数器，每当访问一页时，就把该页对应的计数器加1。操作系统确定一个周期 T，在周期 T 的时间内，若没有发生缺页中断，则把所有的计数器清零，开始一个新的周期重新计数。若在周期 T 的时间内发生了缺页中断，则选择计数值最小的那一页调出(它是最近一段时间内最不常用的页)，同时把所有的计数器清零。这个算法的实现要花很大的开销，并且要确定一个合适的周期 T 也有一定的难度。

6）Clock 置换算法

LRU 算法是较好的一种算法，但由于它要求有较多的硬件支持，故在实际应用中大多采用 LRU 的近似算法。Clock 算法就是用得较多的一种 LRU 近似算法。

(1）简单的 Clock 置换算法

当采用简单 Clock 算法时，只需为每页设置一位访问位，再将内存中的所有页面都通过链接指针链接成一个循环队列。当某页被访问时，其访问位被置1。置换算法在选择以页淘汰时，只需检查页的访问位。如果是0，就选择该页换出；若为1，则重新将它置0，暂不换出，而给该页第二次驻留内存的机会，再按照 FIFO 算法检查下一个页面。当检查到队列中的最后一个页面时，若其访问位仍为1，则再返回到队首去检查第一个页面。图5.23给出了该算法的流程和示例。由于该算法是循环地检查各页面的使用情况，故称为 Clock 算法。但因该算法只有一位访问位，

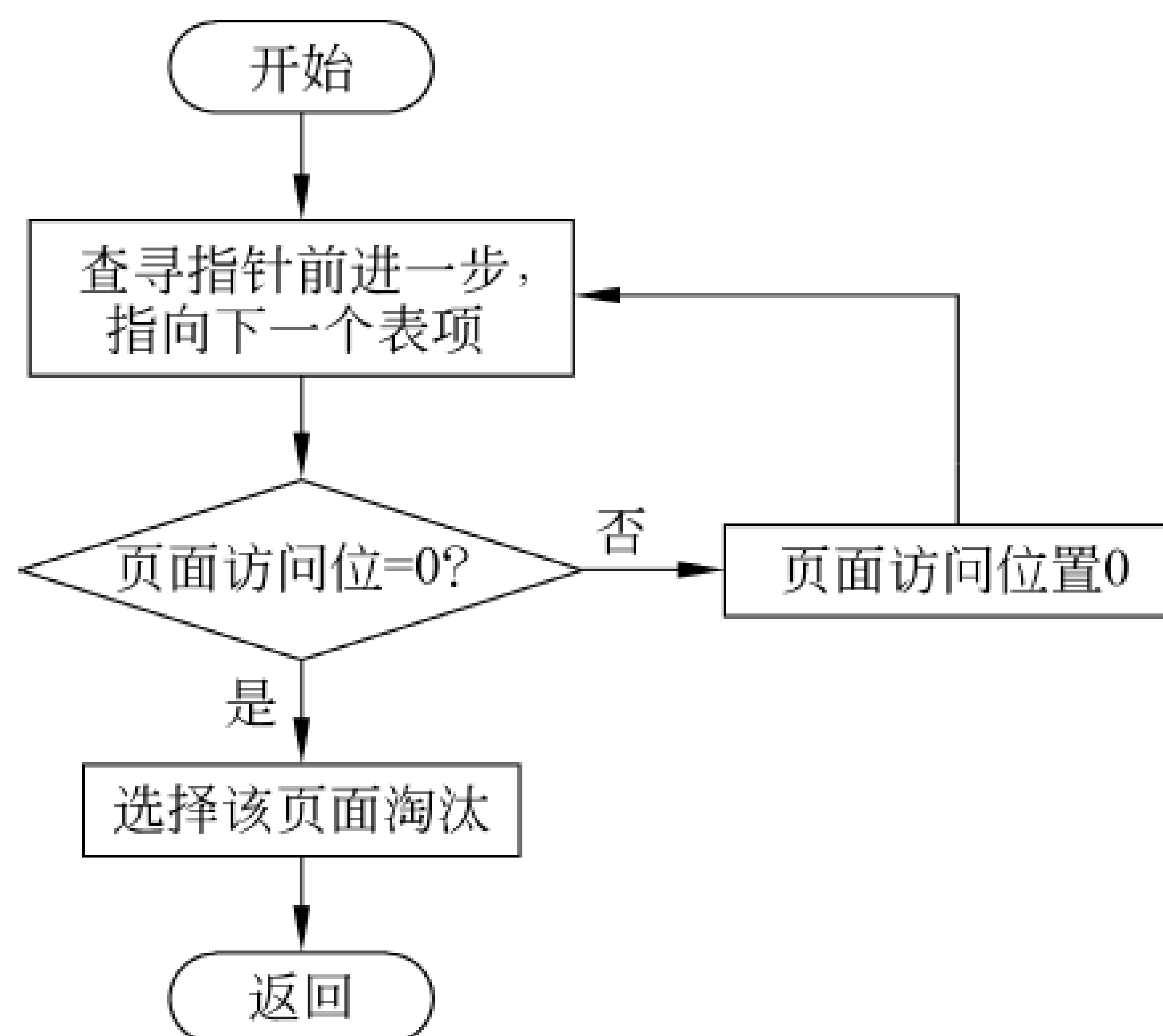


图5.23 简单 Clock 置换算法的流程

只能用它表示该页是否已经使用过，而置换时是将未使用过的页面换出去，故又把该算法称为最近未用算法（Not Recently Used，NRU）。

（2）改进型 Clock 置换算法

在将一个页面换出时，如果该页已被修改过，便须将该页重新写回到磁盘上；但如果该页未被修改过，则不必将它写回磁盘。在改进型 Clock 算法中，除须考虑页面的使用情况外，还须再增加一个因素，即置换代价，这样选择页面换出时，既要是未使用过的页面，又要是未被修改过的页面。把同时满足这两个条件的页面作为首选淘汰的页面。由访问位 A 和修改位 M 可以组合成下面 4 种类型的页面：

1 类（A=0，M=0）：表示该页最近既未被访问，又未被修改，是最佳淘汰页。

2 类（A=0，M=1）：表示该页最近未被访问，但已被修改，并不是很好的淘汰页。

3 类（A=1，M=0）：表示该页最近已被访问，但未被修改，该页有可能再被访问。

4 类（A=1，M=0）：表示该页最近已被访问且被修改，该页可能再被访问。

在内存中的每个页必定是这 4 类页面之一，在进行页面置换时，可采用与简单 Clock 算法相似的算法，其差别在于改进型算法须同时检查访问位与修改位，以确定该页是 4 类页面中的哪一种。其执行过程可分成以下 3 步：

（1）从指针所指示的当前位置开始，扫描循环队列，寻找 A=0 且 M=0 的第一类页面，将所遇到的第一个这类页面作为所选中的淘汰页。在第一次扫描期间不改变访问位 A。

（2）如果步骤(1)失败，即查找一周后未遇到第一类页面，则开始第二轮扫描，寻找 A=0，且 M=1 的第二类页面，将所遇到的第一个这类页面作为淘汰页。在第二轮扫描期间，将所有扫描过的页面的访问位都置 0。

（3）如果步骤(2)也失败，即未找到第二类页面，则将指针返回到开始的位置，并将所有的访问位置 0。然后重复步骤（1），如果仍失败，必要时再重复步骤（2），此时就一定能找到被淘汰的页。

该算法与简单 Clock 算法比较，可减少磁盘的 I/O 操作次数。但为了找到一个可置换的页，可能须经过几轮扫描。换言之，实现该算法本身的开销将有所增加。

假定系统为某进程分配 3 个主存块，开始时为空，依次要访问的页号为 7、0、1、2、0、3、0、4、2、3、0、3、2、1、2、0、1，采用简单 Clock 算法进行页面置换的情况为：先将 7、0、1 三个页面装入内存。访问页面 2 时将发生缺页中断，进行页面置换，按照简单 Clock 算法将把页面 7 淘汰，因为在第一次扫描时，三个页面的访问位都是 1，都被置为 0，第二次扫描时，首先检查到页面 7 的访问位为 0，先被淘汰，页面 2 装入内存，访问位置为 1。下次访问页面 0，在内存，不产生缺页中断，把页面 0 的访问位置为 1。在访问页面 3 时将发生缺页中断，进行页面置换，由于页面 1 的访问位为 0，故页面 1 被淘汰，页面 3 装入内存，访问为置为 1。图 5.24 给出了采用最佳置换算法时的内存页面的置换情况。从图中可以看出共产生了 8 次缺页中断。

7）页面缓冲算法（Page Buffering Algorithm，PBA）

虽然 LRU 和 Clock 置换算法都比 FIFO 算法好，但它们都需要一定的硬件支持，并需付出较多的开销，而且置换一个已修改的页比置换未修改页的开销要大。而页面缓冲算法（PBA）则既可改善分页系统的性能，又可采用一种较简单的置换策略。VAX/VMS 操作系统便是使用页面缓冲算法。它采用了前述的可变分配和局部置换方式，置换算法采用的是 FIFO。该算法规定将一个被淘汰的页放入两个链表中的一个，即如果页面未被修改，就将

访问序列 / FIFO 队列	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0
2		0	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2
3			1	1	1	3	3	3	3	3	3	3	3	1	1	1	1
是否中断	是	是	是	否	否	是	否	是	是	否	是	否	否	是	否	否	否

图5.24　采用简单Clock算法时的内存页面的置换情况

它直接放入空闲链表中；否则，便放入已修改页面的链表中。须注意的是，这时页面在内存中并不做物理上的移动，而只是将页表中的表项移到上述两个链表之一中。

空闲页面链表实际上是一个空闲物理块链表，其中的每个物理块都是空闲的，因此，可在其中装入程序或数据。当需要读入一个页面时，便可利用空闲物理块链表中的第一个物理块来装入该页。当有一个未被修改的页要换出时，实际上并不将它换出内存，而是把该未被修改的页所在的物理块挂在自由页链表的末尾。类似地，在置换一个已修改的页面时，也将其所在的物理块挂在修改页面链表的末尾。利用这种方式可使已被修改的页面和未被修改的页面都仍然保留在内存中。当该进程以后再次访问这些页面时，只需花费较小的开销，使这些页面又返回到该进程的驻留集中。当被修改的页面数目达到一定值时，例如64个页面，再将它们一起写回到磁盘，从而显著地减少了磁盘I/O的操作次数。一个较简单的页面缓冲算法已在MACH操作系统中实现了，只是它没有区分已修改页面和未修改页面。

8）最佳置换算法(Optimal Replacement Algorithm，OPT)

一个理想的调度算法是当要装入一个新页面而必须调出一个页面时，所选择的调出页应该是以后再也不使用的页或者是距当前最长时间以后才使用的页。这种调度算法能使缺页中断率最低。然而，因为谁也无法对程序执行中要使用的页面作出精确的断言，因此这种算法是无法实现的，不过，这个理论上的算法可作为衡量各种具体算法的标准，这个算法被称为"最佳置换算法"。

假定系统为某进程分配3个主存块，开始时为空，依次要访问的页号为7、0、1、2、0、3、0、4、2、3、0、3、2、1、2、0、1，采用最佳置换算法进行页面置换的情况为：先将7、0、1三个页面装入内存。访问页面2时将发生缺页中断，进行页面置换，按照最佳置换算法将把页面7淘汰，因为页面0将在第5次被访问，页面1将在第14次被访问，而页面7以后不被访问。下次访问页面0，在内存，不产生缺页中断。在访问页面3时，由于页面0、1、2在内存，并且这3个页面在以后的访问中页面1是最晚一个要被再访问的，所以页面1被淘汰。图5.25给出了采用最佳置换算法时的内存页面的置换情况，从图中可以看出共产生了7次缺页中断。

3．页面调度算法的进一步说明

假定一个作业或进程共有 n 页，系统分配给它的主存块是 m 块(m，n 均为正整数，且 $1 \leqslant m \leqslant n$)。因此，该作业或进程最多有 m 页可同时被装入主存。如果作业或进程执行中访问页面总次数为 A，其中有 F 次访问的页面尚未装入主存，故产生了 F 次缺页中断。现定义

$$f = F/A$$

把 f 称为缺页中断率。

访问序列 / 主存块	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
1	7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2		0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0
3			1	1	1	3	3	3	3	3	3	3	3	1	1	1	1
是否中断	是	是	是	否	否	是	否	是	否	否	是	否	否	是	否	否	否

图 5.25　采用最佳置换算法时的内存页面的置换情况

1）FIFO 算法的进一步说明

FIFO 算法简单，易实现。可以把装入主存储器的那些页的页号按进入的先后次序排成队列，每次总是调出队首的页，当装入一个新页后，把新页的页号排入队尾。图 5.26 是先进先出调度算法的示例。

访问次序 / 主存块	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
1	•7	•7	•7	2	2	2	•2	4	4	•4	0	0	0	0	•0	•0	•0
2		0	0	•0	•0	3	3	•3	2	2	•2	•2	•2	1	1	1	1
3			1	1	1	•1	0	0	•0	3	3	3	3	•3	2	2	2

图 5.26　先进先出调度算法的示例

这里指针 K 是循环指针，分配给该作业或进程的主存块数是 m 块，即页号队列中有 m 个页号，每次调出一页后，执行 $K=(K+1) \bmod m$ 使得 K 指示的位置为队首，而新装入的那页所在的位置就成了队尾。例如，依次要访问的页号为 7、0、1、2、0、3、0、4、2、3、0、3、2、1、2、0、1，现在分配 3 个主存块，开始时为空。执行时按 FIFO 算法进行页面调度，将产生 12 次缺页中断，缺页中断率为 12/17＝70.5％，页面装入和调出的情况如图 5.26 所示（前面带点的页号为最先进入内存的页）。

FIFO 方法除主存利用率低的缺点外，还可能产生 Belady 现象。

一般说，对于一个进程或作业，如果分配的内存块数越接近它所需要的块数，则发生的缺页次数越少。Belady 现象是：在未给作业分配满足它所需要的主存块数时，则会出现当分配的块数增多时，缺页次数反而增多的现象。例如，依次要访问的页号为 1、2、3、4、1、2、5、1、2、3、4、5，现在分配 3 个主存块，开始时为空。执行时按 FIFO 算法进行页面调度，将产生 9 次缺页中断，缺页中断率为 9/15＝75％，页面装入和调出的情况如图 5.27 所示。

访问次序 / 主存块	1	2	3	4	1	2	5	1	2	3	4	5
1	•1	•1	•1	4	4	•4	5	5	5	5	•5	•5
2		2	2	•2	1	1	•1	•1	•1	3	3	3
3			3	3	•3	2	2	2	2	•2	4	4

图 5.27　分配 3 个主存块时页面装入和调出的情况

对于该例，如果分配4个主存块，开始时为空。执行时按FIFO算法进行页面调度，将产生10次缺页中断，缺页中断率为10/15＝83.3％，页面装入和调出的情况如图5.28所示。

主存块 \ 访问次序	1	2	3	4	1	2	5	1	2	3	4	5
1	•1	•1	•1	•1	•1	•1	5	5	5	•5	4	4
2		2	2	2	2	2	•2	1	1	1	•1	5
3			3	3	3	3	3	•3	2	2	2	•2
4				4	4	4	4	4	•4	3	3	3

图5.28　分配4个主存块时页面装入和调出的情况

FIFO产生Belady现象的原因在于它根本没有考虑程序执行的动态特征。

2）LRU算法的进一步讨论

LRU算法总是选择距现在最长时间内没有被访问过的页面先调出。实现这种算法的一种方法是在页表中为每一页增加一个“计时”标志，记录该页面自上次被访问以来所经历的时间，每被访问一次都应从“0”开始重新计时。于是，要装入新页而产生缺页中断时，检查页表中各页的计时标志，从中选出计时值最大的那一页调出，并且把各页的计时标志全部置“0”，重新计时。当再一次产生缺页中断时，又可找到最近最少使用的页，将其调出。这种实现方法必须对每一页的访问情况时时刻刻地加以记录和更新，实现起来比较麻烦，而且开销大。所以，有的系统还经常使用一种LRU近似算法。

LRU还可用栈的方法实现，这种方法能准确地选择最近最少用的页。栈中存放当前在主存中的页，每当访问一页时就调整一次，使栈顶总是指示最近访问的页，而栈底是最近最少用的页。当发生缺页中断时总选择栈底所指示的页面调出。还是用FIFO算法中的例子看LRU算法的调度情况，如图5.29所示。

主存块 \ 访问次序	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
1	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
2		7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0
3			•7	•0	•1	•2	•2	•3	•0	•4	•2	•4	•0	•3	•3	•1	•2

图5.29　LRU算法的调度示例

设最初主存块为空，在执行中产生缺页中断时，用LRU调度算法选择调出的页面，则总共产生11次缺页中断，缺页中断率为11/17＝64.7％。

4. 影响缺页中断率的因素分析

影响缺页中断率的因素如下。

1）分配给作业或进程的主存块数

一般情况下，分配给作业或进程的主存块数多，则同时装入主存的页面数就多，减少了缺页中断的次数，也就降低了缺页中断率；反之，缺页中断率就高。

从原理上说，每个作业或进程只要能得到一块主存空间就可以开始执行了，但系统的效率太低。根据实验分析，对一共有 n 页的作业或进程来说，只要能分到 $n/2$ 块主存空间时才把它装入主存执行，那么，可使系统获得最高效率。

2）页面的大小

页面的大小取决于主存分块的大小，块大则页面也大，每个页面大了，则作业或进程的页面数就少，装入一页的信息量就大，就减少了缺页中断的次数，降低了缺页中断率；反之，若页面小，则缺页中断率就高。

3）程序编制方法

一个作业或进程怎样编制程序也是值得探讨的，程序编制的方法不同，对缺页中断的次数有很大影响。

例如，有一个程序要把 128×128 的数组置初值“0”，数组中的每个元素为一个字。现假定页面的尺寸为每页 128 个字，数组中的每一行元素存放在一页中。能供这个程序使用的主存块只有一块，开始时把第一页装入了主存。若程序如下编制：

```
int A[128][128],i,j;
  for(j=1;j<128;j++)
    for(i=1;i<128;i++)
      A[i][j]=0;
```

由于程序是按列把数组中的元素清零的，所以，每执行一次 a[i][j]=0 就会产生一次缺页中断。因为开始时第一页已在主存了，故程序执行时就可对元素 A[0][0]清零，但下一个元素 a[1][0]不在该页中，就产生缺页中断。程序按上述的编制方法，每装入一页只对一个元素清零后就要产生缺页中断，于是总共要产生 128×128－1 次缺页中断。

如果程序重新编制如下：

```
int A[128][128],i,j;
  for(i=1;i<128;i++)
    for(j=1;j<128;j++)
        A[i][j]=0;
```

那么，每装入一页后就对一行元素全部清零后才产生缺页中断，故总共只产生 128－1 次缺页中断。

可见，缺页中断率与程序的局部化程度密切相关。一般说，希望编制的程序能经常集中在几个页面上进行访问，以减少缺页中断率。

4）页面调度算法

如何选择调出的页面是很重要的，如果采用了一个不合适的算法，就会出现这样的现象：刚被调出的页面又立即要用，因而又要把它重新装入，而装入不久又被选中调出，调出不久又被重新装入，如此反复，使调度非常频繁。这种现象称为“抖动”或“颠簸”。一个好的调度算法应减少和避免抖动现象，因而页面调度算法对缺页中断率的影响也很大。据估计，采用 FIFO 调度算法产生的缺页中断率约为最佳淘汰算法的 3 倍。

5.4.4 页式存储管理的优缺点

页式存储管理具有如下优点：

(1) 由于它不要求作业或进程的程序段和数据在内存中连续存放,从而有效地解决了碎片问题。

(2) 动态页式存储管理提供了内存和外存统一管理的虚存实现方式,使用户可以利用的存储空间大大增加。这既提高了主存的利用率,又有利于组织多道程序执行。

其主要缺点如下:

(1) 要求有相应的硬件支持。例如地址变换机构、缺页中断的产生和选择淘汰页面等都要求有相应的硬件支持。

(2) 增加了系统开销,例如缺页中断处理等。

(3) 页面调度算法如选择不当,有可能产生抖动现象。

(4) 虽然消除了碎片,但每个作业或进程的最后一页内总有一部分空间得不到利用。如果页面较大,则这一部分的损失仍然较大。

(5) 由于是以块为单位分配存储,而不是以逻辑段为单位分配内存,故程序的共享的实现较为困难。

例 假设一个分页存储管理系统中具有快表,多数活动页表项都可以存在其中,如果页表存放在内存中,内存访问时间是 1μs。若快表的命中率为 85%,则有效访问时间是多少?若快表的命中率为 50%,则有效访问时间是多少?

解:有效访问时间是指通过逻辑地址访问对应物理地址中的数据所花的时间。有快表时,先查找快表(由于速度很快,所花时间忽略不计),若找到了对应的页表项,取出物理块号并拼成物理地址,再访问内存,只需访问内存 1 次;若在快表中没有找到,再在页表中查找,需要访问内存 2 次。

若快表的命中率为 85%,则有效访问时间$=2\times1\mu s+0-1\mu s\times85\%=1.15\mu s$;

若快表的命中率为 50%,则有效访问时间$=2\times1\mu s+0-1\mu s\times50\%=1.5\mu s$。

由于快表的访问时间相对很短,若题目中没有给出快表访问时间,通常可以看成快表访问时间为 0。

5.5 段式和段页式存储管理

分区存储管理要求作业或进程放在连续的存储区中,分页存储管理克服了分区存储管理的这一缺点,但它的分页不是按逻辑意义进行的。为了按逻辑意义上的段进行存储管理,引进了段式存储管理和段页式存储管理。

5.5.1 段式存储管理

段式存储管理方式是在作业或进程装入主存时每个独立段(逻辑段)连续地存放在一个连续的内存区中。段式存储管理支持用户的分段观点,以段为单位进行存储空间的分配。

1. 段式存储管理的基本原理和使用的数据结构

1) 基本原理

每个作业或进程由若干逻辑段组成,每一段都可独立地编制程序,每一段逻辑地址都是从“0”开始,段内的地址是连续的,而段与段之间的地址可以是不连续的。

段式存储管理提供给用户编程时使用的逻辑地址由两部分组成:段号和段内地址,其

格式如图5.30所示。

段号	段内地址

图5.30　段式存储管理时使用的逻辑地址的格式

当地址结构确定以后,允许作业或进程的最多段数及每段的最大长度也就确定了。假定地址用 m 个二进制位表示,其中段内地址占用了 n 位。那么,每个作业或进程最多可分 2^{m-n} 段,每段的最大长度可达 2^n 个字节。从表面上看,段式存储管理的地址结构与页式存储管理的地址结构类似,但是,它们之间有实质上的不同。页式存储管理提供连续的逻辑地址,由系统自动地进行分页;段式存储管理中作业或进程的分段是由用户决定的,每段独立编址,因此,段间的逻辑地址是不连续的。因此段式存储管理中逻辑地址是二维的。

2) 使用的数据结构

在段式存储管理中使用的数据结构主要有段表、请求表、已分配表和未分配表。

(1) 段表

段表用来记录作业或进程的各段在主存中的位置。操作系统在每装入一个作业或进程时为其建立一张段表。

简单的段表由段号、段始址和段长度组成。根据需要在段表中可以增设表项,如权限标志位、内外存标志位、修改位、外存始址和共享标志等。

(2) 请求表

请求表用来记录作业或进程申请或占用内存情况。它记录每个作业或进程段表起始地址和长度,以进行内存分配和地址变换。请求表是整个系统一张,如图5.31所示。

进程号	请求主存量	段表始址	段表长度	状态
…	…	…	…	已分配/未分配

图5.31　请求表示例

(3) 已分配表

已分配表用来记录主存的分配情况,整个系统使用一张表。主要由主存始址、长度、占据该区域的作业名或进程号组成。其格式和作用与分区存储管理方式相同。

(4) 未分配表

未分配表记录内存空闲区状态,整个系统使用一张表。未分配表主要有两种实现方法:空闲块表和空闲块链。其格式和作用与分区存储管理方式相同。

2. 静态段式存储管理

静态段式存储管理是在作业或进程开始执行之前,把该作业或进程的程序段或数据段全部装入内存,每一段放在连续的存储区中,通过数据结构和地址转换机构来实现相应的管理。

1) 主存空间的分配和回收

静态段式存储管理分配主存空间的方法与可变分区管理方式的分配方法相同,有相同结构的主存空间分配表。所不同的是段式存储管理是为作业或进程的每一个逻辑段分配一个连续的主存空间。进行主存空间分配时,根据段长找出一个可容纳该段的一个空闲区,分割这个空闲区,一部分用来装入该段信息,在已分配表和该作业或进程的段表中登记;另一部分仍为空闲区,在未分配表中登记。当没有一个足够大的空闲区时,仍可采用移动技术来

合并分散的空闲区。当把作业或进程装入主存储器后,该作业或进程的段表也就建立起来了。为了提高访问速度,可以把一部分段表组织在高速缓存中形成快表。

作业或进程执行结束时,要收回该作业或进程各段所占用的主存区域,使其成为空闲区,回收存储空间的方法与可变分区管理方式相同。

2) 地址转换与存储保护

段式存储管理要有硬件的地址转换机构作支撑,地址转换过程类似于可变分区方式的地址转换,段表的表目起到了基址/限长寄存器的作用。每执行一条指令时,地址转换机构按逻辑地址中的段号查段表,得到该段对应的表目。当逻辑地址中的段内地址不超过表目中设置的长度,则把表目中的起始地址与段内地址相加,就得到欲访问的主存绝对地址。如果段内地址超过了限定的长度,则产生一个“地址越界”程序性中断事件而暂停作业或进程的执行。地址转换过程如图 5.32 所示。

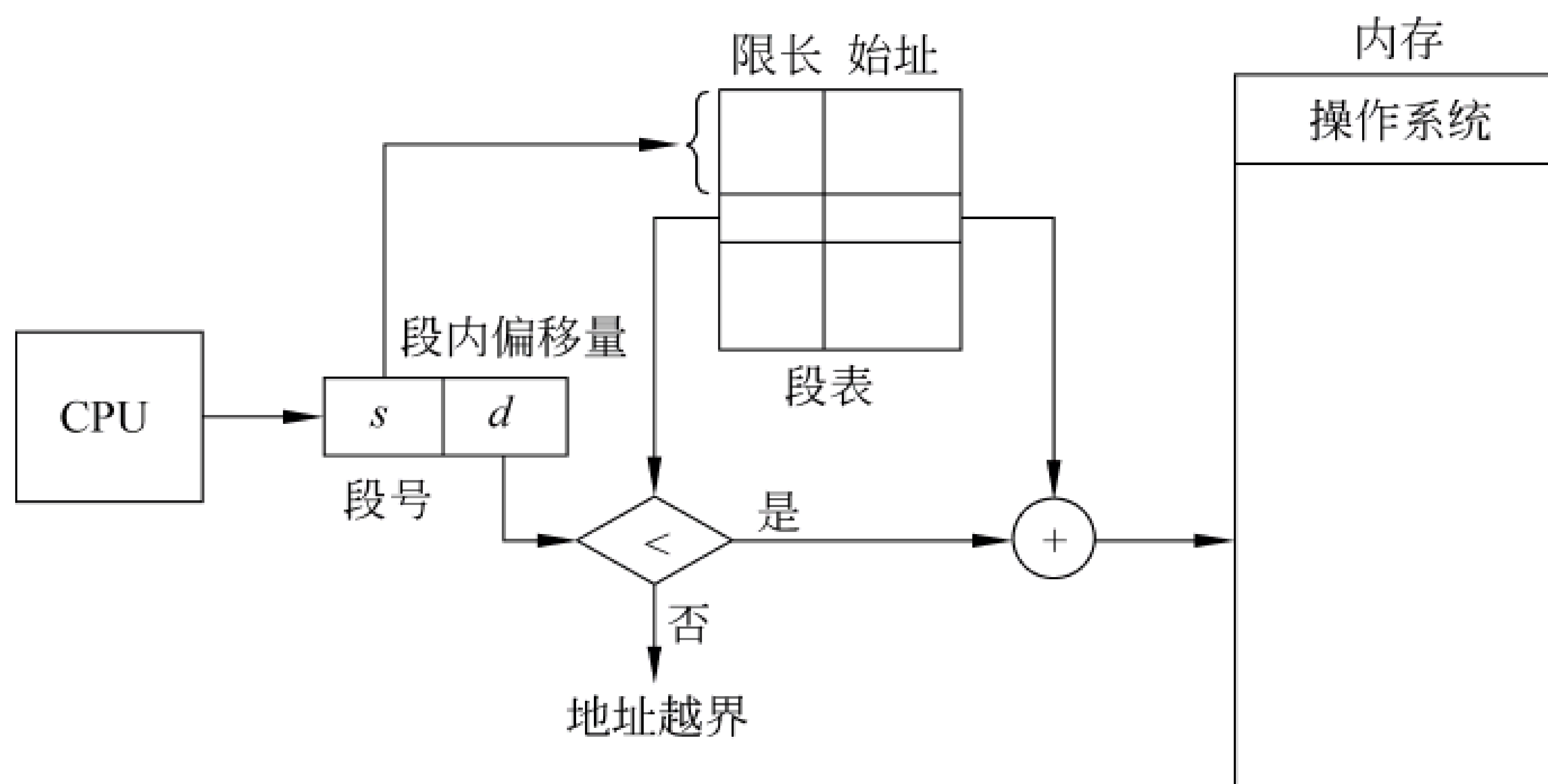


图 5.32 段式存储管理的地址转换

在段式存储管理的多道程序设计系统中,某道作业或进程占用处理器时,地址转换机构就要根据该作业或进程的段表进行地址转换。为此,硬件设置了一个段表控制寄存器,包括基址寄存器和限长寄存器,用来存放当前占用处理器的作业或进程的段表始址和长度。地址转换机构总是按段表控制寄存器的指示找到当前作业或进程的段表,然后进行地址转换。

3) 主存信息共享和保护

由于静态段式存储管理是按逻辑段进行空间分配和管理的,因而十分便于主存信息的共享。此时段表中应包含共享标志和权限标志。如果某段是共享段,则段表中对应条目的共享标志置为“1”,即可共享。对共享段应设置权限标志,以说明该作业或进程对该段的访问权限。

3. 动态段式存储管理

动态段式存储管理是以静态段式存储管理为基础,为用户提供比主存实际容量大得多的虚拟空间。

在动态段式存储管理方式中,段表中应设该段是否在主存的标志、是否修改标志以及各段在磁盘上的位置,已在主存的段仍要指出该段在主存中的起始地址和占用主存区长度。

动态段式存储管理把作业或进程中的各个逻辑段信息都保留在磁盘上,当作业或进程可以投入执行时,首先把当前需要的一段或几段装入主存。作业或进程执行时,若要访问的段已在主存,则按静态段式存储管理中的方式进行地址转换;若要访问的段不在主存,则产

生一个“缺段中断”,由操作系统处理这个中断。处理的办法是,查主存分配表,找出一个足够大的连续区以容纳该逻辑段,如果找不到足够大的连续区则检查空闲区的总和,若空闲区总和能满足该段要求,那么进行适当移动将分散的空闲区集中;若空闲区总和不能满足该段要求,可把主存中的一段或几段调出(如果这些段被修改,应写回磁盘中对应的位置上)使空闲区满足待装入段的需要,然后把当前要访问的段装入主存。段被移动、调出和装入后都要对段表、已分配表和未分配表中的相应表目作修改。新的段被装入后,应让作业或进程重新执行被中断的指令,这时就能找到要访问的段,可以继续执行下去。

动态段式存储管理中的其他问题与静态段式存储管理基本相同。

4. 段式存储管理的优缺点

与页式管理和分区式管理比较,段式管理的优点如下:

(1) 和动态页式管理一样,段式管理也提供了内外存统一管理的虚存技术。与页式管理不同的是,段式虚存每次交换的是一段有完整逻辑意义的信息,而不是像页式虚存那样只交换固定大小的页,从而需要多次缺页中断才能把所需信息完整地调入内存。

(2) 在段式管理中,段长可根据需要动态增长。这对那些需要不断增加或吸收新数据的段来说,将是非常有好处的。

(3) 便于对具有完整逻辑功能的信息段进行共享。

(4) 便于实现动态链接。由于段式管理是按信息的逻辑意义来划分段,每段对应一个相应的程序模块。因此,可用段名加上段入口地址等方法在执行过程中调入相应的段进行动态链接。当然,段的动态链接需要一定的硬件支持,例如需要链接寄存器存放被链接段的入口地址等。

段式存储管理的缺点主要有以下几点:

(1) 段式管理比其他几种方式要求有更多的硬件支持。

(2) 由于在内存空闲区管理方式上与分区式管理相同,在碎片问题以及为了消除碎片所进行的合并等问题上较分页式管理要差。

(3) 允许段的动态增长也会给系统管理带来一定的难度和开销。

(4) 每个段的长度受内存可用区大小的限制。

(5) 段式管理系统在选择淘汰算法时也必须十分慎重,否则也有可能产生抖动现象。

5.5.2 段页式存储管理

段式存储管理要求逻辑意义上的段连续存放在内存中。如果段太长,内存没有满足需要的连续空间,作业或进程将要等待。为把逻辑意义上的段存放,在内存不连续的区域中,引入了段页式存储管理。

1. 基本原理与使用的数据结构

1) 基本原理

程序的分段结构具有逻辑上清晰的优点,但采用段式存储管理的一个弱点是每段必须占据主存储器的一个连续区域。于是,装入一个分段时,可能要移动已在主存储器中的信息。为克服这个缺点,可兼用分段和分页的方法,构成段页式存储管理。

每个作业或进程仍由用户进行分段,每一段的逻辑地址仍是从“0”开始的一组连续地址。但存储管理不是为每一段分配一个连续的主存空间,而是把每一段再分成若干页面,把

一段的信息按页存放在不必相邻的空闲主存块中。所以，段页式存储管理兼顾了段式在逻辑上的清晰和页式在管理上的方便。

段页式存储管理要求把主存储器预先分成大小相等的许多块，在把一段的信息装入主存时，按块的大小分页，一段信息可被分散存放在若干主存块中。因此，段页式存储管理的逻辑地址的格式如图 5.33 所示。



图 5.33 段页式存储管理逻辑地址的格式

2）使用的数据结构

段页式存储管理使用的主要数据结构有以下 3 个：

（1）段表。为每一个装入主存的作业或进程建立一张段表。段表的长度由作业或进程分段的个数决定，段表中的每一个表目指出本段的页表始址和长度。

（2）页表。对装入主存的作业或进程的每一段建立一张页表，页表的长度由对应段所分的页的个数决定，页表中的每一个表目指出本段的逻辑页号与主存块号的对应关系。另外根据需要还可以包含其他的表项，如权限标志位、内外存标志位、修改位、外存始址和共享标志等。

（3）存储块表。与页式存储管理相同。

2. 静态段页式存储管理

静态段页式存储管理方法是在作业或进程开始执行之前，把该作业或进程的程序段或数据段按页全部装入内存，每一段放在连续的或不连续的存储块中，通过数据结构和地址转换机构来实现相应的管理。

执行指令时，地址机构根据逻辑地址中的段号查段表，得到该段的页表始址，然后根据页号查页表，得到对应的主存块号，由主存块号与逻辑地址中的页内地址形成可访问的绝对地址。如果逻辑地址中的段号超出了段表中的最大段号或者页号超出了该段页表中的最大页号，都要形成“地址越界”的程序性中断事件。

3. 动态段页式存储管理

动态段式管理还是以段为单位分配主存空间，整段地调出、装入，有时还要移动，这些都增加了系统的开销。如果按段页式存储管理的方式，把每一段再分成若干页面，那么，每一段不必占用连续的存储空间，甚至，当主存块不够时，可只将一段中的部分页面装入主存。这种管理方式称为动态段页式存储管理。

动态段页式存储管理结合了动态段式和页式的优点，但增加了设置表格（段表、页表）和查表等开销，一般只在大型计算机系统中采用。

5.6 Linux 的存储管理

Linux 的存储管理由两部分组成，第一部分是物理内存的管理，第二部分是虚拟存储器的管理，主要是进程虚拟地址空间的管理。

5.6.1 物理内存的管理

所有的物理内存一部分由核心内存管理,由内核映像使用;另一部分由虚拟存储子系统管理,主要用在其他内核子系统的内存需求、进程的需求和缓冲需求。核心内存管理必须能够快速响应请求,在尽可能地提高内存利用率的同时减少内存碎片问题。

Linux 核心内存管理采用了基于区域的伙伴系统及 slab 分配器。

1. 物理内存的划分

物理内存是以页帧(page frame)为基本单位,页帧的大小固定,对 Intel CPU 默认为4KB。每个页帧由一个 struct page 结构描述。Linux 2.4 内核支持 NUMA 结构,NUMA 结构的物理内存逻辑上统一编址,但在物理上位于不同位置,从而导致访问所需的时间并不一样。访问时间相同的物理区域称为一个节点。为了保证效率,一般不希望有跨节点操作,并且尽可能使用访问时间短的节点。

每个节点的物理内存根据用途不同又分成不同的区域(zone)。例如 x86,分成如下3个区域。

(1) DMA zone:这部分内存是低于16MB的内存,是DMA方式能够访问的物理内存。在内存分配时,尽可能保留这部分内存以供DMA方式使用。

(2) normal zone:这部分内存是16~896MB之间的内存,直接被内核映射。

(3) highmem zone:高端内存,是超过896MB以上的部分,不能被内核直接映射。

在内存分配时,首先要选定节点,然后要根据区域访问优先级别来决定访问次序。比如为DMA方式分配内存,DMA zone 是唯一符合要求的区域。而通常的内存分配,则可先在 normal zone 进行,如果不能满足要求,则在 DMA zone 尝试。

2. 空闲块的管理及分配回收

1) 空闲块的管理

Linux 对空闲块的管理采用两种数据结构:链表和位示图。拥有连续空闲块数(必须是 $2^n, n=0,1,2,\cdots$),相同的区域链成一个链,再利用数组 free_area 记录每个链的头指针。这样的管理体系称为伙伴系统,可以解决“外部碎片”问题。位示图(map)记录每一物理页的状态。

两组连续页面块被认为是一对“伙伴”,必须满足如下条件:

(1) 大小相同,比如说都有 b 个页面。

(2) 物理空间上连续。

(3) 位于后面那个块的最后页面号必须是 $2b$ 的倍数。

图5.34说明了 Linux 物理内存的组织。

图中 free_area 数组的元素0包含一个空闲页(空闲页的编号是0);元素2包含两个以4页为大小的空闲页块,第一个空闲块的起始页编号是3,第二个空闲块的起始页编号是34。

2) 内存的分配与回收

以8个页块为例,说明它的分配和回收过程。8个页块,则可以用4个链表来描述页块的空闲情况,分别描述1、2、4、8个连续页块的情况。设开始均为空闲,则只有第3个链表不为空(从0编号),表示页块[1~8]均空闲。

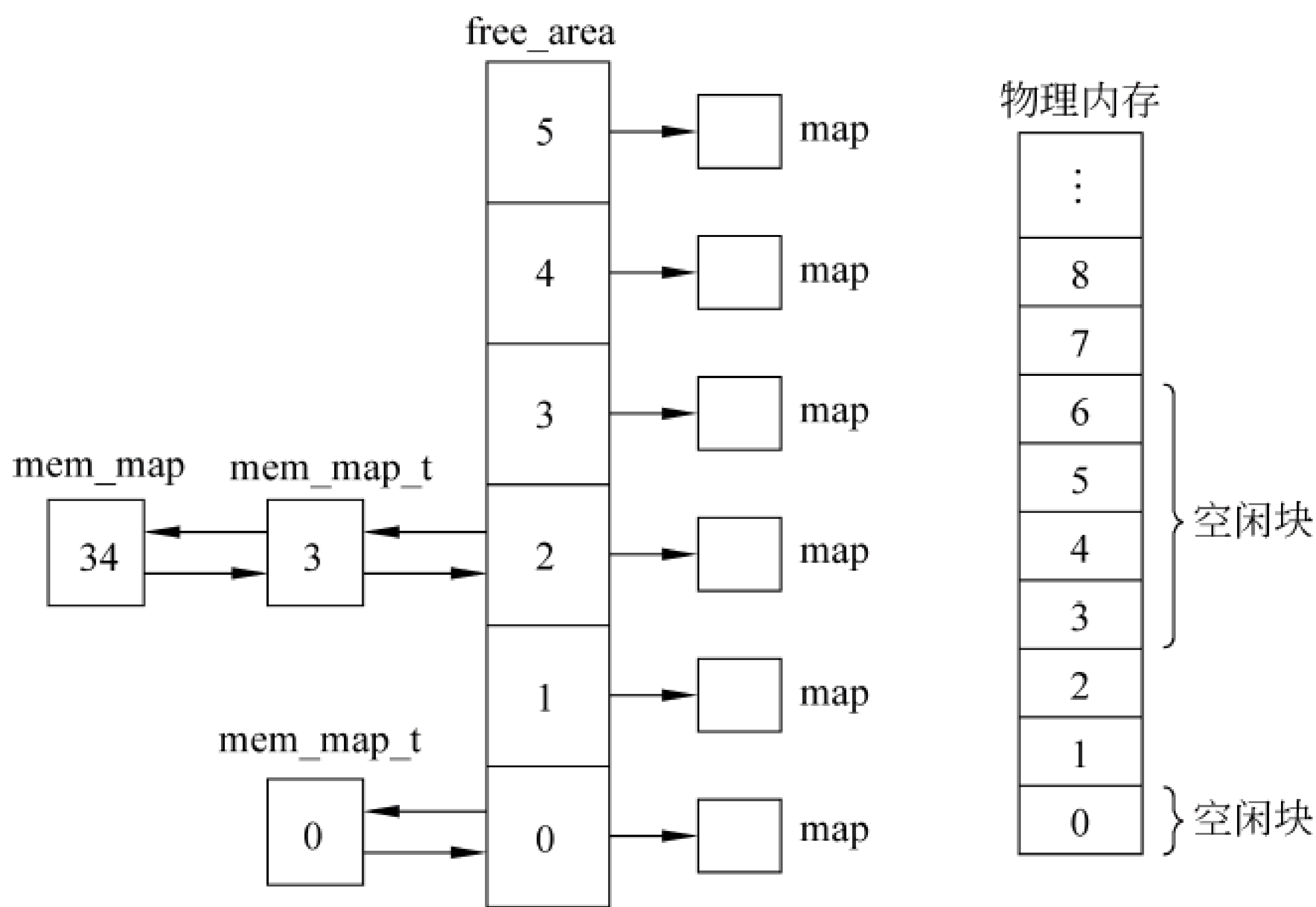


图 5.34 Linux 物理内存的组织示意图

现在申请 2 个页块,先在第 1 个链表中找,发现为空,再到第 2 个链表中找,仍为空,再向上找,直到找第 3 个链表,发现有 8 个连续空闲页块。清空链表 3 的对应项,并把 8 个页块分为两半,[5～8]加入链表 2,[1～4]继续分为两半,[3～4]加入链表 1,页块[1～2]用来满足申请。假设继续申请 4 个页块,先到链表 2 查找页块[5～8],则将这 4 个页块分配出去,同时清空链表 2 的对应项。

如果用户释放页块[1～2],系统并不将其插入链表 1,而是看它的伙伴[3～4]是否空闲,如果是则表明可组成一个更大的连续页块[1～4]。所以此时将[3～4]从链表 1 中摘除。继续查看空闲页块[1～4]的伙伴[5～8]是否空闲,[5～8]已被分配,表明不可能合成更大的连续空闲页块,此时才将页块[1～4]加入链表 2。

从上面可以看到,分配页块时尽量动用小的连续页块,回收页块时则尽可能将空闲的伙伴合成大的连续页块,从而在很大程度上解决了内存的“外部碎片”问题。

3. slab 分配器

伙伴系统以页帧为基本分配单位,但使用物理空间小于页的情况很多,如果分配一页则浪费了存储空间。为此 Linux 引入了 Sun OS 操作系统中的技术——基于伙伴系统的 slab 分配器。

slab 分配器的基本思想是,为经常使用的小对象建立缓冲,小对象的申请与释放都通过 slab 分配器来管理。这样做的优点在于:其一,充分利用了空间,减少了内部碎片;其二,管理局部化,尽可能少地与伙伴系统打交道,从而提高了效率。

slab 分配器为不同的常用对象生成不同的缓冲,每个缓冲存储相同类型的对象。但某种对象的缓冲区并非由各个对象直接构成,而是由一连串的 slab 构成,每个 slab 又由一个或多个连续的物理页帧组成,包含了若干同种类型的对象。

除了上面讨论的特定对象的缓冲外,Linux 还提供了 13 种通用的缓冲,其存储对象的单位大小分别为 32B、64B、128B、256B、512B、1KB、2KB、4KB、8KB、16KB、32KB、64KB 和 128KB。这些缓冲用来满足特定对象之外的普通内存需求。单位的大小呈级数增长,保证了内部碎片率不超过 50%。

5.6.2 进程空间的管理

每个进程空间通过进程的页目录和页表实现与物理内存间的映射。进程需要空间时并不开始分配物理内存，而是分配一块虚拟空间，直到真正需要对物理内存进行操作时才通过请求调入页面机制分配物理内存。虚拟内存以页为基本单位，大小与物理页帧相等。

1. 页表机制

页表机制与硬件的体系结构密切相关，下面以386体系结构为例，描述如何把进程空间的线性地址转换成为物理地址。386系列既支持分段机制，也支持分页机制，Linux主要采用了分页机制，常规情况页的大小为4KB，页面可以映射到任一物理页帧。386下进程的线性地址为32位，分为如图5.35所示的3个部分。

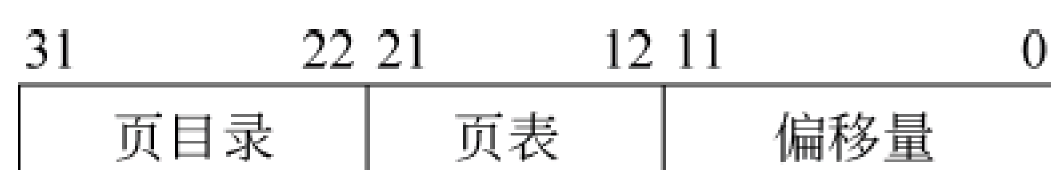


图 5.35　386下进程的线性地址

其中：

(1) 页目录：记录在页目录中的索引。

(2) 页表：记录在页表中的索引。

(3) 偏移量：用来表示在页帧中的偏移。

每个进程都有一个页目录，当进程运行时，寄存器CR3指向该页目录的基址。图5.36显示了从线性地址转换成物理地址的过程。

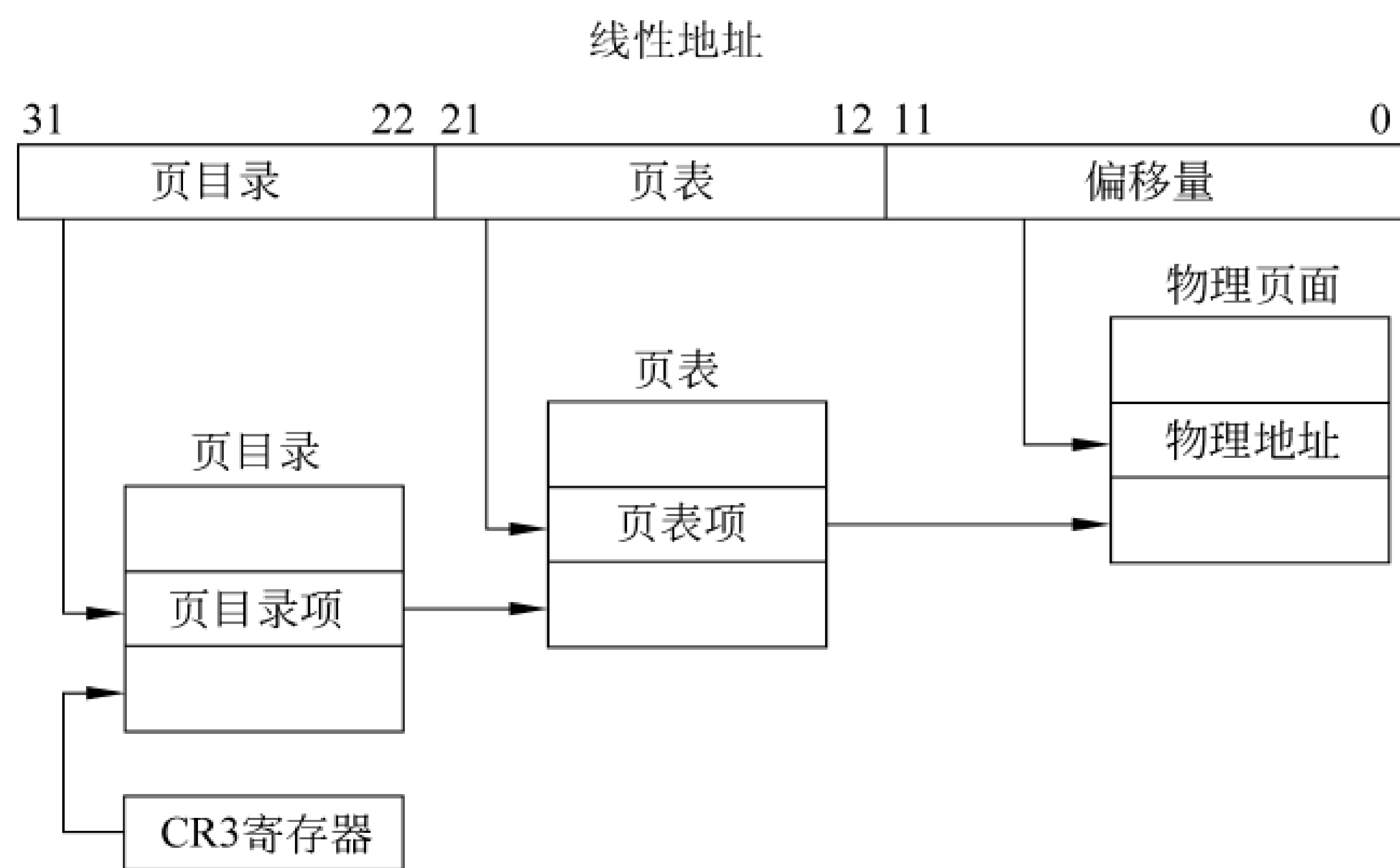


图 5.36　线性地址到物理地址的转换

采用两级分页的好处是节省了存储空间。两级分页对于32位机器是合适的，对于64位机器，采用三级分页则较为合理。

2. 进程空间的管理

Linux对进程虚拟空间的管理采用了请求页式技术。标准Linux的虚拟页表应为三级页表，依次为页目表(PGD，Page Directory)、中间页表(PMD，Page Middle Directory)和页表(PTE，Page Table)，如图5.37所示。

在Intel微机上，Linux的页表结构实际为两级。80386体系结构的页式管理机制中的页目录是PGD，页表是PTE，而PMD和PGD是合二为一的，对PMD的操作转为对PGD的操作。

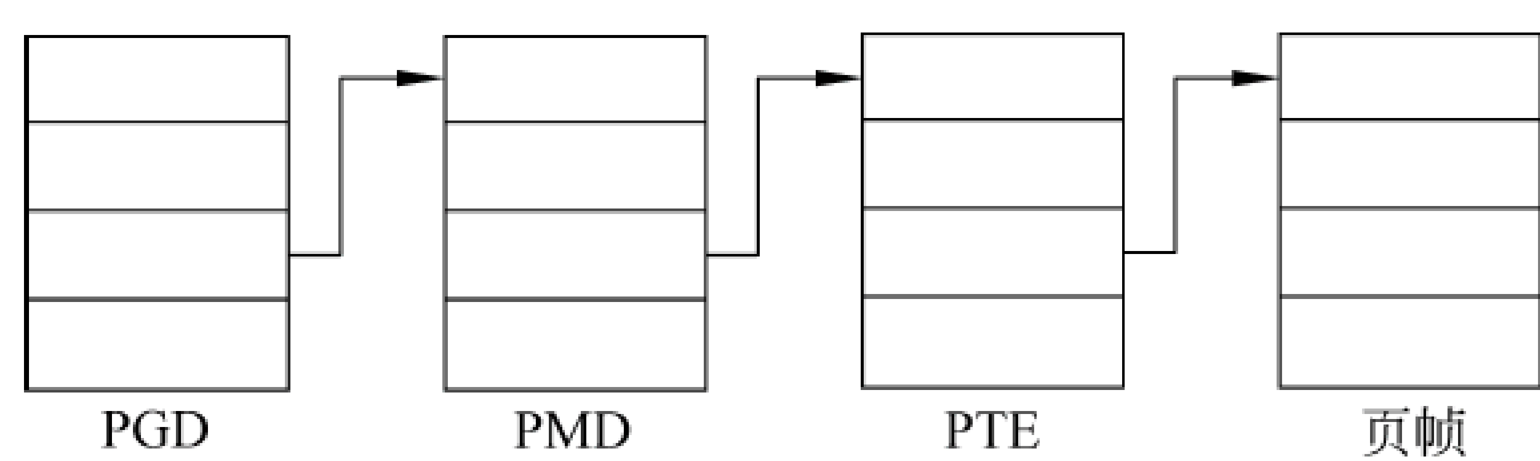


图 5.37 Linux 的三级页表结构

每当启动一个新进程，Linux 都为其分配一个 task_struct 结构，包含内存管理信息，另外它内含一个 mm_struct 结构，此结构包含了用户进程与存储有关的信息：进程的页目录和指向 vm_area_struct 结构的指针。Linux 的进程空间用一系列的 vm_area_struct 结构来描述程序中的不同区域(代码区、数据区等)。每个 vm_area_struct 结构描述进程的一段连续区域，并且在该区域上的访问权限相同，各区域按线性地址的高低次序链接在一起。当区域的数目较多时，将建立一棵平衡二叉树以保证搜索速度。mm_struct 和 vm_area_struct 结构中的详细内容参见有关书籍。图 5.38 是进程虚拟空间管理使用的数据结构及其关系示意图。

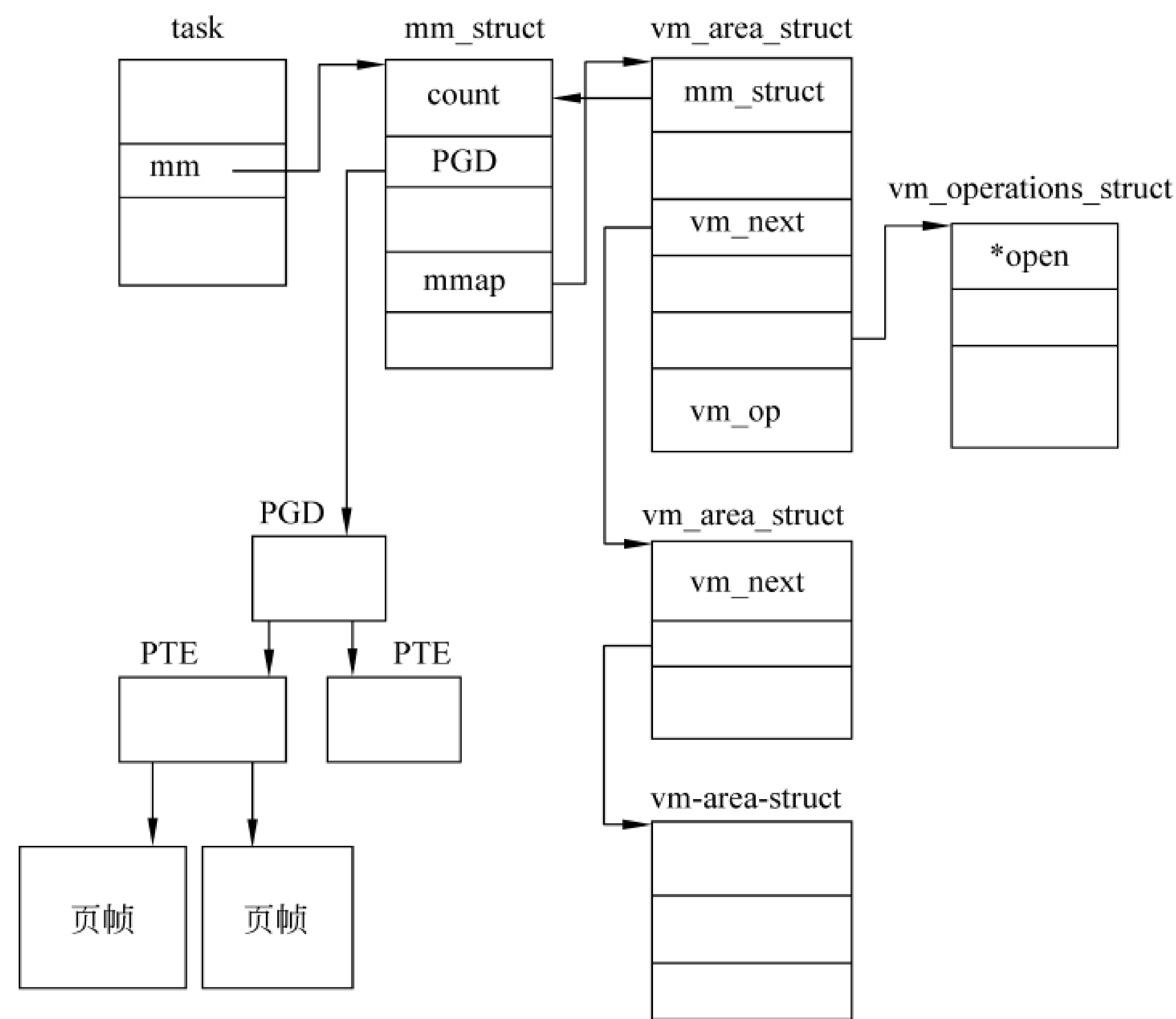


图 5.38 进程虚拟空间管理使用的数据结构及其关系示意图

3. 页面异常的处理

导致页面异常的原因主要有以下两类：

(1) 编程错误。可分为内核程序错误和用户程序错误。

(2) 操作系统故意引发的异常。操作系统合理利用硬件机制在适当时间触发异常，使得该异常的处理程序被调用以达到预期目的。

5.6.3 Linux 虚存的保护

1. 多任务及保护

Linux 允许每个用户最多可运行 256 个任务。Linux 使用了四级保护机制:0 级供操作系统内核使用,1 级供系统调用使用,2 级供共享库使用,3 级供应用程序使用。Linux 内核由系统内的所有任务共享。每个任务有各自的私有代码及数据区,存储在用户空间,因而对系统中的其他任务不可见。

2. 同一任务内的保护

在一个任务之内,定义了 4 种执行特权级别,用来限制对任务中的段进行访问。这些任务,按照包含在段中的数据的敏感性及任务中不同部分的程序,按可信任的程度进行分区。最敏感的数据分配最高的特权级别。特权级别用数字 0~3 表示,数字 0 表示最高特权级别。图 5.39 所示为特权级的典型应用。

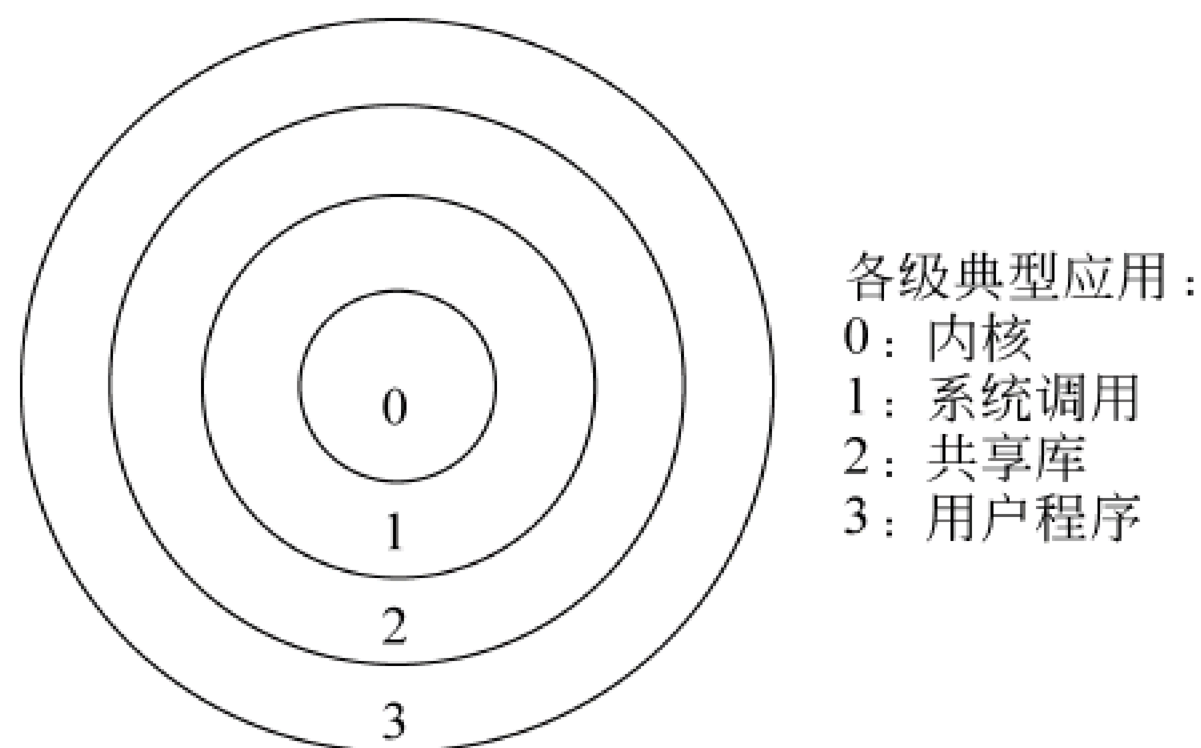


图 5.39 特权级的典型应用

5.7 本章小结

存储器由内存和外存两部分组成。本章主要讨论内存管理问题,主要包括内存的分配和回收、地址转换、内存数据保护与共享和内存扩充等。

覆盖技术和交换技术是内存扩充常用的两种技术,覆盖技术用于一道作业的内存相对扩充,交换技术用于多道作业的内存相对扩充。

分区存储管理是内存管理的主要思想,它是把主存中的用户区作为一个连续区或分成若干个连续区进行管理。当划分多个连续区时可采用固定分区方式或可变分区方式进行管理。

最后介绍了 Linux 的存储分配。主要包括伙伴系统及 slab 分配器、进程空间的管理、页面异常的处理和 Linux 虚存的保护。

各种存储方法的对比如表 5.1 所示。

表 5.1 各种存储方法的对比

<table>
<tr><th rowspan="2">管理方法 / 功能</th><th rowspan="2">单分区</th><th colspan="2">多分区</th><th colspan="2">页式</th><th colspan="2">段式</th><th colspan="2">段页式</th></tr>
<tr><th>固定</th><th>可变</th><th>静态</th><th>动态</th><th>静态</th><th>动态</th><th>静态</th><th>动态</th></tr>
<tr><td>主存分配方式</td><td>静态</td><td>静态</td><td>动态</td><td>静态</td><td>动态</td><td>静态</td><td>动态</td><td>静态</td><td>动态</td></tr>
<tr><td>适用环境</td><td>单道</td><td colspan="2">多道</td><td colspan="2">多道</td><td colspan="2">多道</td><td colspan="2">多道</td></tr>
<tr><td>所用数据结构</td><td></td><td>主存分配表</td><td>已分配表、未分配表(空闲区表和空闲块链)</td><td colspan="2">页表、请求表和存储块表(位示图法和空闲块链法)</td><td colspan="2">段表、请求表、主存分配表和未分配表</td><td colspan="2">段表、页表和存储页面表</td></tr>
</table>

续表

管理方法 功能	单分区	多分区		页式		段式		段页式	
		固定	可变	静态	动态	静态	动态	静态	动态
分配算法/置换算法		分配：顺序	分配：最先适应、最优适应和最坏适应（回收时有合并分区问题）		置换：随机淘汰算法、轮转法、FIFO、LRU、LFU		置换：随机淘汰算法、轮转法、FIFO、LRU、LFU		置换：随机淘汰算法、轮转法、FIFO、LRU、LFU
重定位方式	静态	静态	动态	静态	动态	静态	动态	静态	动态
地址转换公式	绝对地址＝用户区首址＋逻辑地址	绝对地址＝下限寄存器值＋逻辑地址	绝对地址＝基址寄存器值＋逻辑地址	绝对地址＝块号×块长＋页内地址（页表、快表）		绝对地址＝段起始地址＋段内地址（段表、快表）		绝对地址＝块号×块长＋页内地址（段表、页表、快表）	
硬件地址转换机构	不用	用	用	用	用	用	用	用	用
存储保护关系式	界限寄存器值≤绝对地址≤主存最大地址	下限寄存器值≤绝对地址≤上限寄存器值	逻辑地址≤限长寄存器的值	页内地址≤页长，逻辑地址中的页号在页表中	页内地址≤页长	段内地址≤段长，逻辑地址中的段号在段页表中	段内地址≤该段长	静态页式与静态段式结合	动态页式与动态段式结合

习　题

1. 存储管理的功能。

2. 为了有效合理地利用内存，设计内存的分配和回收方法时，必须考虑和确定哪几种策略和数据结构？

3. 什么是地址重定位？有哪几种方式？说明它们的含义。

4. 常用的内存信息保护方法有哪几种？

5. 覆盖技术的实现是基于程序的什么特性？

6. 简述覆盖技术的基本思想、适用场合及优缺点。

7. 说明交换的含义、引入交换的目的、适用的场合及它的优缺点。

8. 简述常用的内存管理方法，说明它们使用的数据结构及作用，并说明分配、回收、地址转换、内存扩充、共享和保护是如何实现的，以及各种方法的优缺点。

9. 在可变分区中常用的分配算法有哪几种？在回收时存在什么问题？是如何解决的？

10. 在页式管理中为何使用快表？在段式和段页式管理中可否使用快表？如果可以使用快表，其中的内容是什么？

11. 如何理解页式管理中的逻辑地址是线性的而段式管理中的逻辑地址是二维的？

12. 为什么有缺页中断问题?常用的页面置换算法有哪几种?影响缺页中断率的因素有哪些?

13. 假定存储器空闲块有如图5.40所示的结构。请你构造一串内存请求序列,对该请求序列最先适应分配算法能满足,而最优适应分配算法则不能。

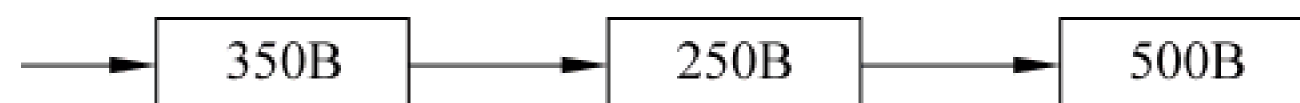


图5.40 存储器空闲块

14. 某操作系统采用动态分区存储管理技术。操作系统在低地址占用了100KB的空间,用户区主存从100KB处开始占用512KB。

初始时,用户区全部为空闲,分配时截取空闲分区的低地址部分作为已分配区。执行以下请求、释放操作序列:请求300KB;请求100KB;释放300KB;请求150KB;请求50KB;请求90KB。回答以下问题:

(1) 采用最先适应算法时,主存中有哪些空闲分区?画出主存分布图,并指出空闲分区的首地址和大小。

(2) 采用最优适应算法时,主存中有哪些空闲分区?画出主存分布图,并指出空闲分区的首地址和大小。

(3) 若随后又要请求80KB,针对上述两种情况产生什么后果?说明为什么。

15. 某系统采用动态页式存储管理。主存每块为256B,现要把一个128×128的二维数组置初始值为"0"。在分页时把数组中的元素每行放在一页中,假定系统只分给用户一页数据区。

(1) 对如下程序段,执行完要产生多少次中断?

```
short a[128][128],i,j;
for(j=0;j<128;j++)
  for(i=0;i<128;i++)
    a[i][j]=0;
```

(2) 为减少缺页中断的次数,请改写上面的程序,使之仍能完成所要求的功能并减少中断次数。在动态页式存储系统中,一个程序依次访问的页面为1,2,3,4,1,2,5,1,2,3,4,5。设分配给该程序的内存页面是m,试分别计算$m=3$和$m=4$时FIFO和LRU两种置换算法的页面中断次数。结果说明了什么?

16. 在动态页式存储管理中,运行一共有8页的作业,且作业在主存中分配到4块主存空间,作业执行时访问的页面顺序为7、0、1、2、0、3、0、4、2、3、0、3、2、1、2、0、1、7、0、1。请问用FIFO和LRU调度算法时:

(1) 它们的缺页中断率分别是多少?要求有过程。

(2) 访问一次内存需要100ns,缺页中断处理时间是25ms,问这20页的平均访问时间是多少?

17. 某计算机的逻辑地址空间和物理地址空间均为64KB,按字节编址。若某进程最多需要6页数据存储空间,页的大小为1KB。操作系统采用固定分配局部置换策略为此进程分4个页面,如表5.2所示。

表 5.2 页表

页号	页框号	装入时间	访问位	页号	页框号	装入时间	访问位
0	7	130	1	2	2	200	1
1	4	230	1	3	9	160	1

当该进程执行到260时刻时，要访问逻辑地址为17CAH的数据，请回答下列问题：

(1) 该逻辑地址对应的页号是多少？

(2) 若采用先进先出(FIFO)置换算法，该逻辑地址对应的物理地址是多少？要求给出计算过程。

(3) 若采用时钟(Clock)置换算法，该逻辑地址对应的物理地址是多少？要求给出计算过程(设搜索下一页的指针沿顺时针方向移动，且当前指向2号页面，如图5.41所示)。

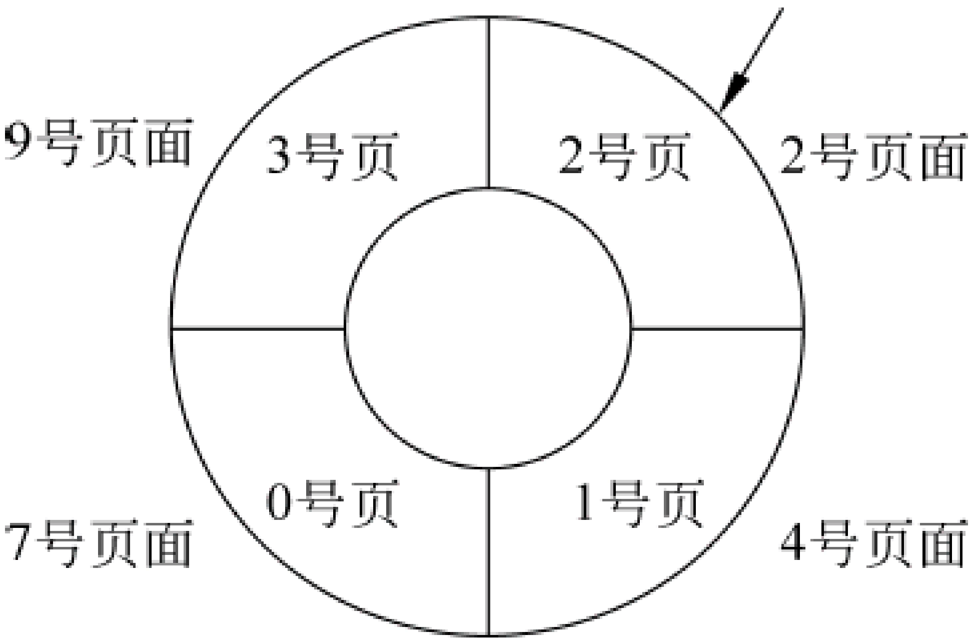


图 5.41 当前指向2号页面

18. 什么是Belady现象？哪些算法可以产生Belady现象？
19. Linux的物理内存采用何种管理方式？
20. Linux解决“外部碎片”的方法是什么？举例说明伙伴算法。
21. 为什么引入slab分配器？
22. Linux是如何实现虚存保护的？

第6章　文件管理

文件系统是操作系统的重要组成部分，它主要对以文件形式存放在外部存储器上的信息进行管理，主要包括如何以文件的形式组织信息，如何通过文件名对存储在存储介质上的文件进行操作，如何实现文件的共享、保护和保密等。如何提高文件系统的可靠性和如何对外部存储空间进行有效管理也是文件系统所要解决的问题。

6.1　文件和文件系统

用户使用计算机来完成自己的某项任务时，会碰到下列问题：

（1）使用现有的软件资源来协助完成自己的任务，这些软件从哪里获得。

（2）编制完成的或未完成的程序存放在什么地方，需要访问的数据又存放在哪里，从而使得人们可以再利用已有的软件资源。

事实上，这两个问题是一个怎样对软件资源（程序和数据）进行透明存取，并能令这些程序和数据做到招之即来的问题。在早期的计算机系统中，由于硬件资源的限制，只能用卡片或纸带来存放程序或数据。这些卡片和纸带都分别编号存放，当用户需要使用它们时，再把这些卡片和纸带放在读卡机上输入计算机系统中。显然，这些人工干预的控制和保存软件资源的方法不可能做到透明存取，这就极大地限制了计算机的处理能力和CPU等计算机硬件资源的利用率。

大容量直接存取的磁盘存储器以及顺序存取的磁带存储器等硬件设备的出现，为程序和数据等软件资源的透明存取提供了物质基础。这导致了对软件资源管理质的飞跃——文件系统的出现。文件系统把相应的程序和数据看作文件，并把它们存放在磁盘或磁带等大容量存储介质上，从而做到对程序和数据的透明存取。这里的“透明存取”是指不必了解文件存放的物理结构和查找方法等与存取介质有关的部分，只需给出代表某个程序或数据的文件名，文件系统就会自动地完成对与给定文件名相对应的文件的有关操作。

6.1.1　文件

通常人们把文件定义为一段程序或数据的集合，这是一种较为模糊的说法。在计算机系统中，文件被解释为一组赋名的相关联字符或者是相关联记录（一个有意义的信息单位）的集合。

文件含义的两种解释定义了两种文件形式。赋名的字符流文件是一种无结构文件或流式文件。目前常用的操作系统，例如Linux和Windows等均采用无结构文件形式。无结构文件由于采用字符流方式，与源程序和目标代码等在形式上是一致的，因此，该方式适用于源程序和目标代码等文件的存储。另一种文件形式是记录式文件，它是由相关记录组成的，其基本信息单位是记录。而记录是由N（$N>1$）个字节组成的具有特定意义的信息单位。记录式文件主要用于信息管理。

在有些操作系统中，从字符流文件的角度出发，设备也被看作是赋予特殊文件名的文件。从而，系统可以对设备和文件实施统一管理，以致大大简化了设备管理程序和文件系统的接口设计。

文件名由用户给定，它是字母、数字以及规定的一些特殊字符组成的一个串，有些系统规定必须是英文字母开始且允许一些其他的符号出现在文件名的非打头部分。例如 C. prj 和 ccdos. lib 均为合法文件名。

6.1.2 文件的分类

对文件进行分类的目的是为了便于管理和控制文件。由于不同系统对文件的管理方式不同，对文件的分类方法也就有很大不同。为了方便系统和用户了解文件的类型，在许多操作系统中都把文件的类型作为文件的扩展名，在文件名和扩展名之间用“. ”隔开。如扩展名为. prj，表示该文件为工程项目类文件；扩展名为. c，表示该文件为 C 语言的源程序文件。

文件可以按各种方法进行分类，常用的分类方式举例如下。

(1) 按用途分类

可把文件分成系统文件、库文件和用户文件。

(2) 按保护级别分类

根据限定的使用文件的权限可分成可执行文件、只读文件和读写文件等。

(3) 按信息流向分类

由物理设备的特性决定了文件信息的流向，一般可分成输入文件、输出文件和输入输出文件。

(4) 按存放时限分类

根据系统保留文件的时间可分成临时文件、永久文件和档案文件。

(5) 按设备类型分类

根据文件存储介质的设备类型可分成磁盘文件、磁带文件、卡片文件和打印文件等。

(6) 按文件的组织结构分类

文件的组织结构是指文件的构造方式。用户和文件系统往往从不同的角度对待同一个文件。用户从使用角度组织文件，把能观察到的且可以处理的信息根据使用要求构造为文件，由用户构造的文件结构称为文件的逻辑结构，对应的文件称为逻辑文件。文件系统要从文件的存储和检索的角度组织文件，文件系统根据用户对文件的存取方式以及存储介质的特性决定以什么样的形式把用户文件存放到存储介质上，在存储介质上的文件构造方式称文件的物理结构，对应的文件称为物理文件。

6.1.3 文件系统

1. 文件系统的概念

操作系统中与管理文件有关的程序和数据的集合称为文件系统，它负责为用户建立、撤销、读写、修改和复制文件，还负责完成对文件的按名存取和进行存取控制，以及对外存储空间的管理。

2. 文件系统的特点

文件系统具有如下特点：

(1) 友好的用户接口,用户只对文件进行操作,而不管文件结构和存放的物理位置。

(2) 对文件实现按名存取,具体操作对用户透明。

(3) 某些文件可以被多个用户或进程所共享。

(4) 文件系统大都使用磁盘、磁带和光盘等大容量存储器作为存储介质,因此,可存储大量信息。

3. 文件系统必须完成的工作

文件系统作为操作系统的一部分,必须能够完成以下工作:

(1) 为了合理地存放文件,必须对磁盘等辅助存储器空间(或称文件空间)进行统一管理。在用户创建新文件时为其分配空闲区,在用户删除或修改某个文件时,回收和调整存储区。

(2) 为了实现按名存取,需要有一个用户可见的文件逻辑结构,用户按照文件逻辑结构所给定的方式进行信息的存取和加工。这种逻辑结构是独立于物理存储设备的。

(3) 为了便于存放和加工信息,文件在存储设备上应按一定的顺序存放。这种存放方式被称为文件的物理结构。

(4) 完成对存放在存储设备上的文件信息的查找。

(5) 完成文件的共享和保护功能。

6.2 文件的逻辑组织

用户按自己对信息的使用要求组织文件,由于这种文件是独立于物理环境而构造的,所以把用户概念中的文件结构称为文件的逻辑结构,对应的文件称为逻辑文件。逻辑文件有如下两种形式:流式文件和记录式文件。

6.2.1 流式文件

流式文件是指用户对文件内的信息不再划分,整个文件依次由字符所组成,是一种无结构文件。例如,源程序文件就是由一串顺序的字符组成的流式文件。对流式文件,用户常常以长度或特殊字符提出读取文件信息的要求。

6.2.2 记录式文件

记录式文件是指用户对文件内的信息按逻辑上独立的含义再划分成基本的信息单位,每个单位称为一个逻辑记录(简称记录)。一个逻辑文件是由若干个逻辑记录组成的,称为记录式文件。记录式文件中的逻辑记录可依次编号,其序号称为逻辑记录号(简称记录号)。例如,某专业的学生成绩管理文件中,每个学生的信息可作为一个逻辑记录,每个逻辑记录由学号、姓名、所在班级、数学、外语和操作系统成绩 5 个数据项组成,如图 6.1 所示。

对记录式文件,逻辑记录是文件内可以独立存取的最小信息单位。当用户请求文件系统读出一个逻辑记录后,用户可以对逻辑记录中的各个数据项进行处理。为了能正确快速地存取逻辑记录,对记录式文件中的每个逻辑记录至少要有一项特殊的信息,利用它可把同一文件中的各个逻辑记录区分开来。把能用来唯一地标识某个逻辑记录的数据项称为记录的主键。在一个记录式文件中,主键相当重要且不可少,但主键不是唯一的。图 6.1 中的

记录号	学号	姓名	数学	外语	操作系统
1	020101	王红	67	79	87
2	020102	张睿	84	86	76
3	020103	李华	60	89	84
4	020104	李小红	87	68	69

图 6.1　记录式文件结构示例

“学号”和“记录号”都可作为主键，根据给定的“学号”或“记录号”都能找出一个特定的逻辑记录。

把逻辑记录中除主键外的其他各个数据项都称做次键，利用次键能快速找出具有某一特性的所有记录。例如，要找出外语成绩在 60 分以上的人，则利用“外语”这个次键，直接比较各记录的外语成绩，很快就可得到结果。可见利用次键能把文件中的信息按需要快速分类。例如，列出外语成绩不及格的人员名单或一班有哪些学生操作系统成绩在 80 分以上，等等。

根据各系统设计的要求不一样，记录既可以是定长的，也可以是变长的。记录的长度可以短到一个字符，也可以长到一个文件，这要由系统设计人员确定。

通常记录式文件有 4 种结构：连续结构、多重结构、转置结构和顺序结构。下面分别介绍这几种结构。

1. 连续结构

连续结构是一种把记录按生成的先后顺序连续排列的逻辑结构。连续结构的特点是适用性强，可用于所有文件（字符流式的无结构文件实质上是记录长度为一个字符的连续结构文件），且记录的排列顺序与记录的内容无关。这有利于记录的追加与变更。但是连续结构文件的搜索性能较差，例如，要找出某个指定键的记录时，系统必须对整个文件进行搜索。

2. 多重结构

如果把记录按键和记录名排列成行列式结构，则一个包含 n 个记录名、m 个（$m \leqslant n$）个键的文件构成一个 $m \times n$ 维行列式（见图 6.2）。其中，如果第 $i(1 \leqslant i \leqslant m)$ 行和第 $j(1 \leqslant j \leqslant n)$ 列所对应的位置上为 1，则表示键 K_i 在记录 R_j 中；对应位置上为 0，则表示键 K_i 不在记录 R_j 中。同一个键也可以同时属于不同的记录。

如果按行列式结构来排列记录，将会浪费较多的存储空间，为此，把行列式中那些为零的项去掉，并以 K_i 为队首，以包含键 K_i 的记录为队列元素来构成一个记录队列。对于一个有 m 个键的队列来说，这样的队列有 m 个，这 m 个队列构成了该文件的多重结构（multi_list），如图 6.3 所示。

$$
\begin{array}{c} \\ K_1 \\ K_2 \\ \vdots \\ K_m \end{array}
\begin{array}{c} \begin{array}{cccc} R_1 & R_2 & \cdots & R_n \end{array} \\ \begin{bmatrix} 1 & 0 & \cdots & 1 \\ 0 & 0 & & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 0 \end{bmatrix} \end{array}
$$

图 6.2　文件的记录名和键构成的行列式

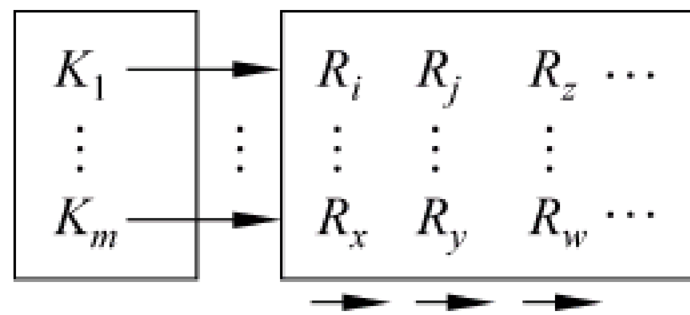


图 6.3　文件的多重结构

3. 转置结构

在图 6.3 的多重结构中，每个队列中和键直接相连的只有一个记录。这种结构虽然在搜索时要优于连续结构，但在搜索某一特定记录时，必须在找到该记录所对应的键之后，再在该键所对应的队列中顺序查找。与此相反，转置结构(inverted life)把含有相同键的记录指针全都指向该键，也就是说，把所有与同一键对应的记录的指针连续地置于目录中该键的位置下面，如图 6.4 所示。转置结构最适合于给定键后的记录搜索。

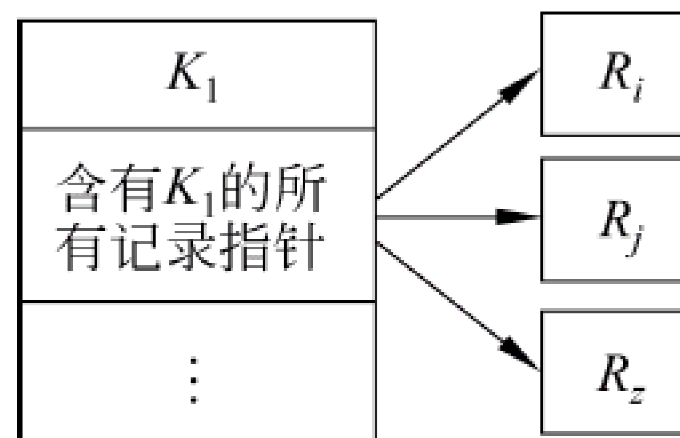


图 6.4 文件的转置结构

4. 顺序结构

如果系统要求按某种优先顺序来搜索或追加、删除记录，则最好采用顺序结构。如果给定了顺序规定(例如按字母顺序)，则把文件中的键按规定的顺序排列起来就形成了顺序结构文件。

6.2.3 存取方法

用户通过对文件的存取来完成对文件的修改、追加和搜索等操作。常用的存取方法有 3 种：顺序存取法、随机存取法(直接存取法)和按键存取法。

顺序存取是按照文件的逻辑地址顺序存取。在记录式文件中，这反映为按记录的排列顺序来存取，例如，若当前读取的记录为 R_i，则下一次读取的记录被自动地确定为 R_i 的下一个相邻的记录 R_{i+1}。在无结构的字符流文件中，顺序存取反映当前读/写指针的变化。在存取完一段信息后，读/写指针自动加上或减去该段信息的长度，以便指出下次存取时的位置。

随机存取法允许用户根据记录的编号来存取文件的任一记录，或者是根据存取命令直接把读写指针移到欲读/写处来对文件进行读写。

UNIX、Linux 和 MS-DOS 系统都采用顺序存取和随机存取两种方法。

按键存取法是一种用在复杂文件系统，特别是数据库系统中的存取方法。文件的存取是根据给定的键或记录名进行的。按键存取法首先搜索到要进行存取的记录的逻辑位置，再将其转换到相应的物理地址后进行存取。

按键存取法对文件的搜索包括两种：键的搜索和记录的搜索。对键的搜索是在用户给定所要搜索的键名和记录之后，确定该键名在文件中的位置；而记录的搜索则是在搜索到所要查找的键之后，在含有该键的所有记录中查找出所需要的记录。显然，对于不同的逻辑结构的文件，其搜索方法和搜索效率都是不一样的。

对键或记录的搜索与其他数据搜索问题一样，都属于表格搜索问题(table lookup)。有许多搜索算法用来解决表格搜索问题，这些算法可以大致分为 3 种类型：线性搜索法(linear search algorithm)、散列搜索法(hash coding algorithm)和二分搜索法(binary search algorithm)。

1. 线性搜索法

线性搜索法是一种最简单、最直观的搜索方法。它从第一个键或记录开始，依次和所要搜索的键或记录相比较，直到找到所需要的记录为止。线性搜索法所需要的搜索时间与所搜索的表格大小的 1/2 成正比。这是因为找到一个所需要的记录平均要和表中登记的总项数的 1/2 项比较后才能得到。

线性搜索法的搜索效率较低，在文件中记录个数较多时不宜采用。

2. 散列搜索法

散列搜索法被广泛用于现代操作系统的数据查找。散列搜索法的核心思想是定义一个散列函数 $h(k)$，使得对于给定的键 k，散列函数 $h(k)$将其变换为 k 所对应的地址。

在使用散列函数进行搜索时，有时会出现两个不同的输入值变换到同一地址的问题。即对于 $k_1 \neq k_2$，有 $h(k_1)=h(k_2)=A$。显然，k_1 和 k_2 中，至少有一个与 A 中的内容不一致。也就是说，由散列变换得到的结果并不是所要搜索的键。这种问题称为散列冲突。解决散列冲突的方法是采用多次散列探索、使用随机函数和采用平方散列函数等方法。

3. 二分搜索法

二分搜索法用于在按某个键值排好顺序的记录中查找给定的记录。首先将要查找的键值与文件中间位置处记录的键值相比较，根据它们之间的关系决定下一步是到前半部分还是下半部分中继续查找。对于顺序结构排列的键或记录来说，二分搜索法具有较高的搜索效率。

二分搜索法的好处是搜索效率高。与线性搜索法相比，当 n(表长)=16 时，它比线性搜索法约快 2 倍；当 $n=1024$ 时，其平均搜索速度要快 50 倍。不过，二分搜索法需要事先把搜索对象按一定顺序排列，故也需要一些额外时间。

6.3 文件的物理组织

用户的逻辑文件要存放到存储介质上时，文件系统要根据存储设备的类型和用户采用的存取方式决定文件在存储介质上的组织方式。组织在存储介质上的文件依赖于物理的存储设备和物理的存储空间，可以看作是相关的物理块的集合，所以，把文件在存储介质上的组织方式称为文件的物理结构，对应的文件称为物理文件。在现代计算机系统中，磁盘和磁带被广泛使用，现以记录式文件在磁盘和磁带上的组织结构来介绍文件的物理组织的有关内容。

6.3.1 磁带文件的组织

磁带机是一种顺序存取的设备，因此，组织在磁带上的文件都是采用顺序结构。将一个文件中在逻辑上连续的信息存放到存储介质的依次相邻的块上，便形成顺序结构，把顺序结构的文件称为顺序文件。文件存放在磁带上时，可按图 6.5 的形式组织。

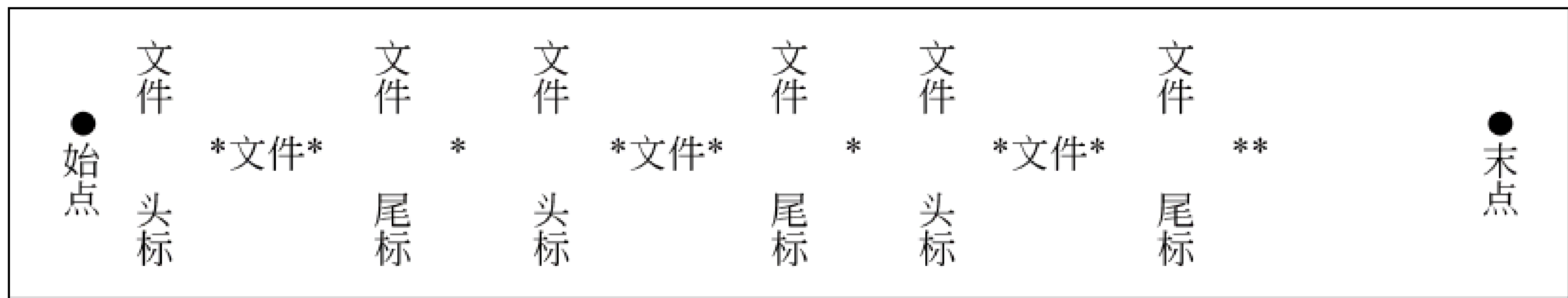


图 6.5 磁带文件的一种组织形式

磁带上的每个文件都有文件头标、文件信息和文件尾标 3 个组成部分。

1. 文件头标

文件头标用来标识一个特定的文件和说明文件的属性，文件头标的内容可以有用户名、文件名、文件的分块数和块的长度等。

2. 文件信息

这是用户逻辑文件中的信息,可把这些信息存放在若干块中,这些块中信息的顺序与逻辑文件中的信息顺序一致。

3. 文件尾标

文件尾标用来表示一个特定的文件信息结束。

其中文件头标和文件尾标是文件系统管理磁带文件时的控制信息。当用户要读磁带上的一个指定文件时,文件系统从磁带的始点开始搜索,先读出第一个文件头标,比较用户名和文件名,若是用户指定的文件,则可读出随后的文件信息;若不是用户指定的文件,则让磁头前进到下一个文件头标的位置,然后读出该文件头标再进行比较,直到找到指定的文件。如果比较到最后一个文件头标,仍不是用户所指定的文件,则表示所要找的文件不在这卷磁带上。

为了能方便、快速地检索磁带文件,在磁带上的各类信息之间用一个称为"带标"的特殊字符(在图6.5中用*表示)将其隔开,最后用两个带标表示磁带上的有效信息结束。磁带机工作时能识别带标,当文件系统读出一个文件头标且确认是用户指定的文件,则只要让磁带机"前进一个带标",磁头就可停在指定文件的开始位置。之后,用户就可请求文件系统顺序读出文件信息。如果在读文件信息时磁带机识别到一个带标,则磁带机立即暂停工作。这时,操作系统的处理程序继续读出磁带上的下一块信息,当读出的是文件尾标时,表示文件信息结束,用户的读请求已超过了文件长度,提示用户不能再读。同时,让磁带机再前进一个带标,使磁头停在下一个文件头标的位置。当文件系统读出一个文件头标且判定不是用户指定的文件,则只要让磁带机"前进三个带标",磁头就可快速地定位到下一个文件头标,然后继续搜索比较。如果不是欲读文件的文件头标,但磁带机却识别到带标,这表示在某文件尾标后连续出现了两个带标,此时磁带上已无有效文件,对该卷磁带的搜索工作可结束。

磁带上的文件适合在顺序存取方式下使用,除了对文件信息顺序存取外,如果对文件也是顺序依次使用,则可省去来回倒带搜索的时间。否则的话,每次都要把磁带退回到始点,从第一个文件开始搜索比较以找出指定的文件。

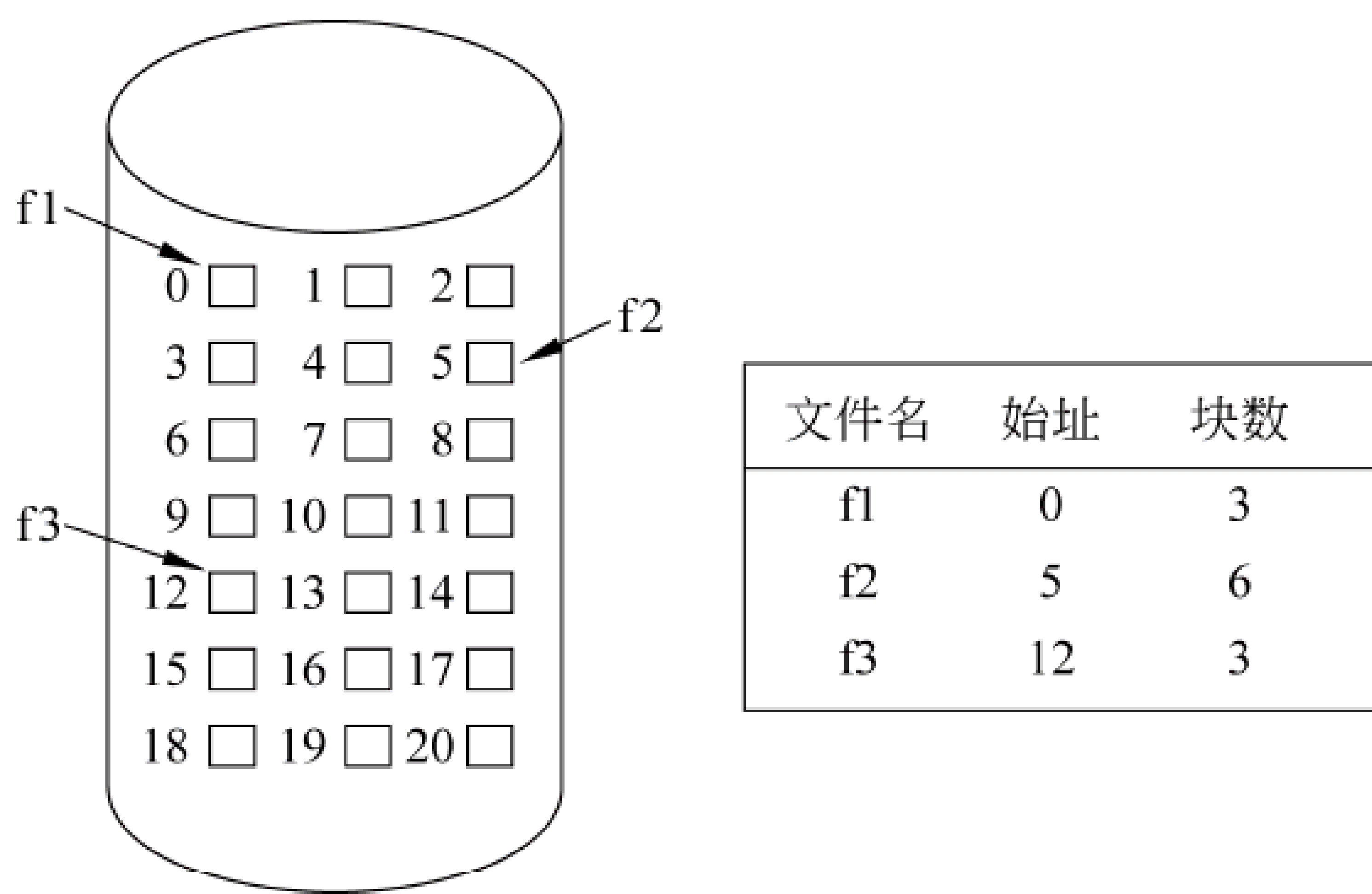
6.3.2 磁盘文件的组织

文件在磁盘上可以有多种组织方式,常用的组织方式有顺序结构、链接结构和索引结构。

1. 顺序结构

一个文件在逻辑上连续的信息被存放到磁盘上依次相邻的物理存储块上,这是一种逻辑记录顺序与磁盘物理块的顺序相一致的文件结构,把这种顺序结构的文件称为顺序文件或连续文件。

通常,若用户总是以逻辑记录的先后次序使用文件,即当前访问第i个记录,则下一次一定访问第$i+1$个记录,那么该文件就可采用顺序结构组织在磁盘上。首先,按文件中逻辑记录的个数确定应占的磁盘块数,找出能存放文件的连续空闲块。然后把文件信息存放到这些磁盘块中并建立文件目录。目录中应指出文件名、文件在磁盘上的起始块号以及占用的块数,如图6.6所示。



文件名	始址	块数
f1	0	3
f2	5	6
f3	12	3

图 6.6　磁盘文件顺序结构示例

对顺序存取的文件采用顺序结构的最大优点是存取速度快，只要记住当前访问的逻辑记录所在的块号，然后连续访问下一个物理块内的记录即可。因此，不必每次都去查找记录的存放位置，减少了检索时间。

顺序结构也存在一些问题：

(1) 磁盘存储空间的利用率不高。磁盘上的每个顺序文件都占用连续的磁盘块，当某个文件被撤销，则它所占的磁盘空间应归还成为空闲块，归还的空间可用来存储其他文件。由于老文件不断地被撤销，新文件不断地被存储进来，使的连续的磁盘空间被分散化了。对一些连续块数较小的空闲块，可能满足不了文件的需要而无法利用。

(2) 对输出文件很难估计需多少磁盘块。在一般情况下，作业执行的结果不是一次形成的，而是边处理边产生结果。因此，对类似这样的输出(结果)文件很难预先估计长度，也就难以确定应分配多少个连续的磁盘块。

(3) 影响文件的扩展。如果一个顺序文件需要扩展，则一定要有与其相邻的空闲磁盘块用以顺序存放被扩展的信息。但是，与其相邻的磁盘块很可能已被其他文件占用，使文件的扩展受到影响。

为了克服上述问题，有的文件系统对顺序结构的文件采用如下附加措施：

(1) 要存储一个文件时，先分配若干连续的磁盘块，把文件信息顺序存放在这些块中，如果存储空间不够，则再分配一组连续的磁盘块，两个连续空间之间加一个链接指针。此时文件的物理结构已不是严格的顺序结构了。

(2) 把一个文件划分成几个能独立存取的顺序子文件，这样，各个顺序子文件只要占用相对较少的连续磁盘块，容易得到满足。由于各顺序子程序是可独立存取的，所以，经这样划分后的文件结构本质上仍是顺序文件。

2. 链接结构

把逻辑文件中的各个逻辑记录任意存放到一些磁盘块中，这些磁盘块可以分散在磁盘的任意位置。这样顺序的逻辑记录被放在一些不连续的、不具备顺序关系的磁盘块上。如果用指针把这些磁盘块按逻辑记录的顺序链接起来，则形成了文件的链接结构，以链接形式组织的文件称为链接文件或串联文件。

链接结构的特点是每个磁盘块中都要有一个指针指向链接文件中的下一个磁盘块，最

后一块中的指针可用特殊字符(例如“-1”)表示文件到此结束。把链接文件中的第一个磁盘块号和最后一块的块号登记到文件目录中以便查找,如图 6.7 所示。

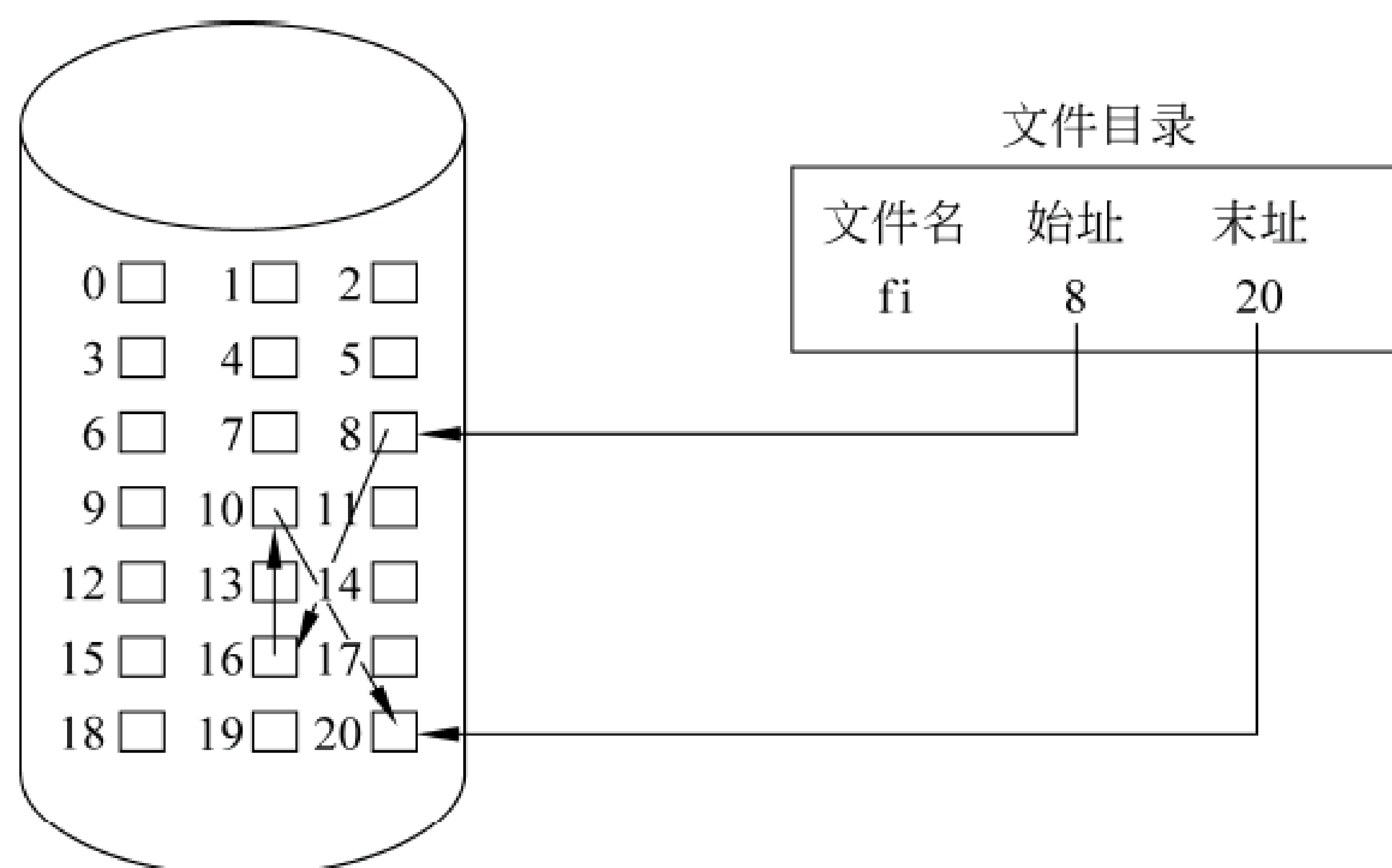


图 6.7 磁盘文件链接结构示例

文件的链接结构解决了顺序结构中的所有问题,磁盘上的所有空闲块都可以被利用,建立文件时也不必事先考虑文件的长度,只要有空闲的磁盘块,文件就可继续扩展。此外,还可根据需要在文件的任何位置插入一个记录或删除一个记录。对文件进行扩展、插入记录时,先寻找磁盘上的空闲块用以存放文件信息,然后修改相应的链接指针,使新的信息链接到文件的适当位置。如果要删除一个记录,也要修改链接指针,把该记录所占的磁盘块从链接文件中脱离出来,然后把该磁盘块置为空闲块并登记在文件系统中。

文件按链接结构组织后,除第一条逻辑记录在磁盘上的位置可以从文件目录中得到外,其余各条记录只能在把前面的记录逐条读出之后才能得到。比如,想要得到第 i 条记录的信息,则必须从文件的首块开始,跟随指针依次读出前面的 $i-1$ 条记录,才能得到第 i 条记录的存放位置,然后再去读出第 i 条记录的信息。以后若又要第 j 条记录的信息,则还需再从头搜索。所以对链接文件采用顺序存取方式是高效的,采用随机存取方式将是低效的。

对于链接结构的文件还应该注意如下几个问题:

(1) 每一个磁盘块中既存放了文件信息,又存放了用于管理的指针,因而浪费了一定的存储空间。有的系统把多个磁盘块组成一个“簇”,以簇为单位分配存储空间,以减少指针占用的空间。

(2) 读写磁盘上的信息以块为单位,当读出一块信息后应把其中的指针分离出来,仅把属于逻辑文件的信息传送给用户,以保证用户使用文件信息的正确性。

(3) 在存取文件时,如果某个指针丢失或被破坏,则错误的指针可能指向其他文件而导致混乱。为了提高可靠性,有的系统采用双指针或在每个磁盘块中再加入文件名。这样以牺牲空间来换取可靠性。

3. 索引结构

索引结构是实现文件非连续存储的另一种方法,索引结构为每个文件建立一张“索引表”,把指示每个逻辑记录存放位置的指针集中存放在索引表中。通常,把索引表保存在某一个磁盘块中,文件目录中指出索引表的存放位置。采用索引结构的文件称“索引文件”。如图 6.8 所示。

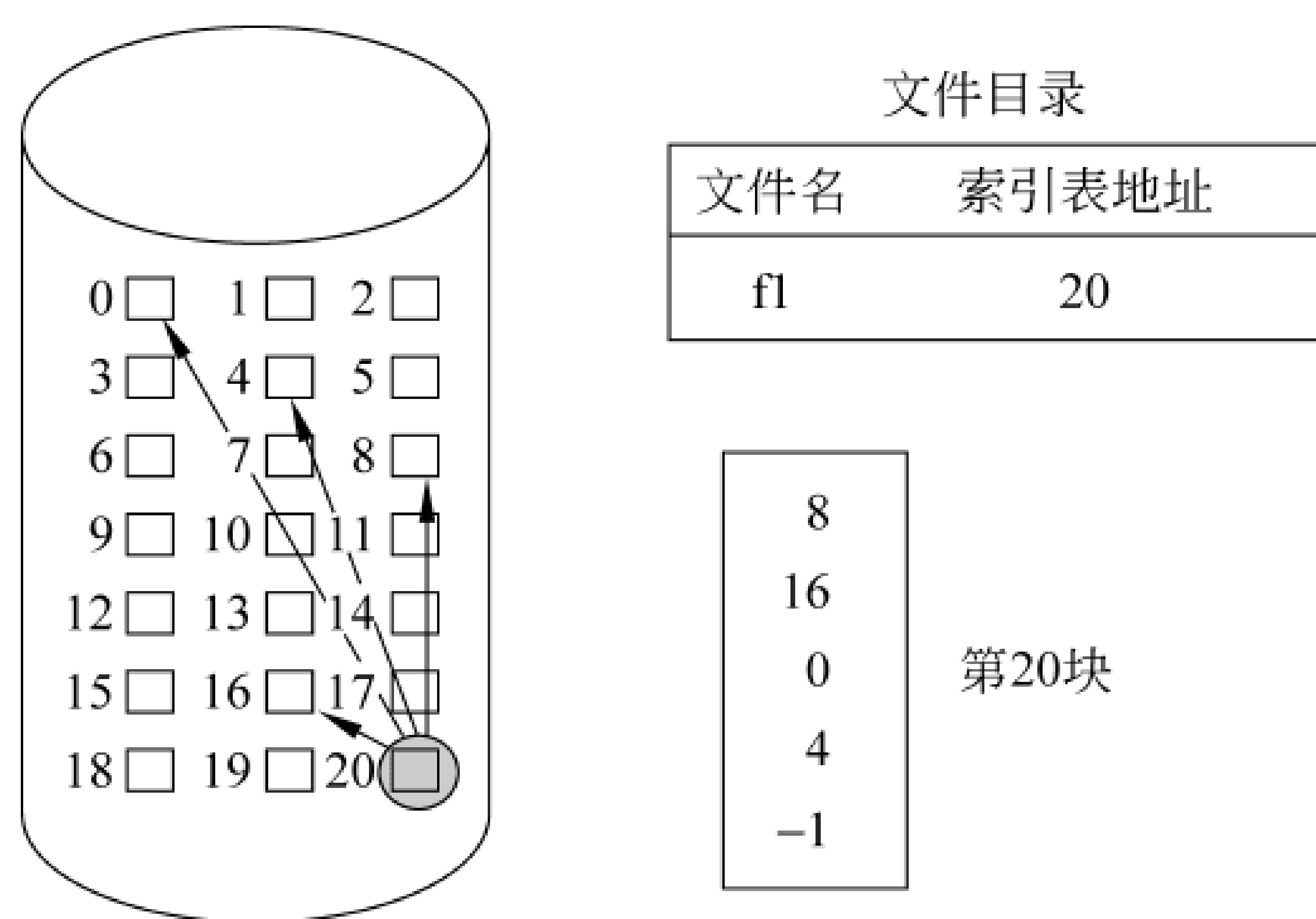


图 6.8 磁盘文件的索引结构示例

索引表中的每个登记项指出一个逻辑记录的存放位置,各个磁盘块号可以按与之对应的逻辑记录的顺序登记在索引表中。这样,第 i 个登记项指示了第 i 条逻辑记录所在位置。当索引表中的登记项数大于逻辑记录个数时,可用特殊字符(比如"—1")表示无效登记项。

存取文件时,先检查该文件的索引表是否已读入主存,如果不在主存中,则根据文件目录的指示将索引表读入主存。对索引文件既可采用顺序存取方式,又可采用随机存取方式。当顺序存取时,只要顺序检索索引表中的登记项,就可按各记录存放的位置依次读出逻辑记录。当随机存取时,对于给定的记录号,例如,请求读第 i 条记录,根据索引表在主存中的起始地址立即可找到第 i 个登记项,按照登记项中指针指示的地址就可以读出第 i 条逻辑记录。

对索引文件能方便地实现文件的扩展、记录的插入和删除。对索引文件进行修改后,它的索引表也被修改了,为了在以后使用文件时不出差错,应把修改过的索引表写回磁盘覆盖原来的索引表。

由于索引结构既适合顺序存取记录,又可方便地按任意次序随机存取记录,且容易实现记录的增、删和插入,所以索引结构被广泛应用。但是,采用索引结构额外增加了索引表占用的空间开销和读写索引表的时间开销。

当一个文件中记录很多时,索引表就很庞大,有时要用多个磁盘块存放一个文件的索引表,可把存放索引表的各磁盘块用指针链接起来。因此,当随机存取某个记录时,可能要沿链搜索才能找到该记录的存放地址,这是很费时间的。

Linux 操作系统对索引表作了精心的设计,采用多级索引的结构。共有 15 项,它的前 11 项可看成一级指针,直接存放文件数据所在的磁盘块号。数组的第 13 项是一个二级指针,指向的磁盘块并不包含文件的数据,而是一系列的一级指针,这些一级指针才用来指向磁盘块。数组的第 14、15 项分别是三级指针和四级指针,访问数据方式可由第 13 项类推。这种方法保证了对大量的小文件访问效率高,同时又支持大文件,如图 6.9 所示。

例 6.1 某文件系统以硬盘作为文件存储器,物理块大小为 512B。有文件 A 包含 590 个逻辑记录,每个记录占 255B,每个物理块存放 2 个记录。文件 A 在该文件目录中的

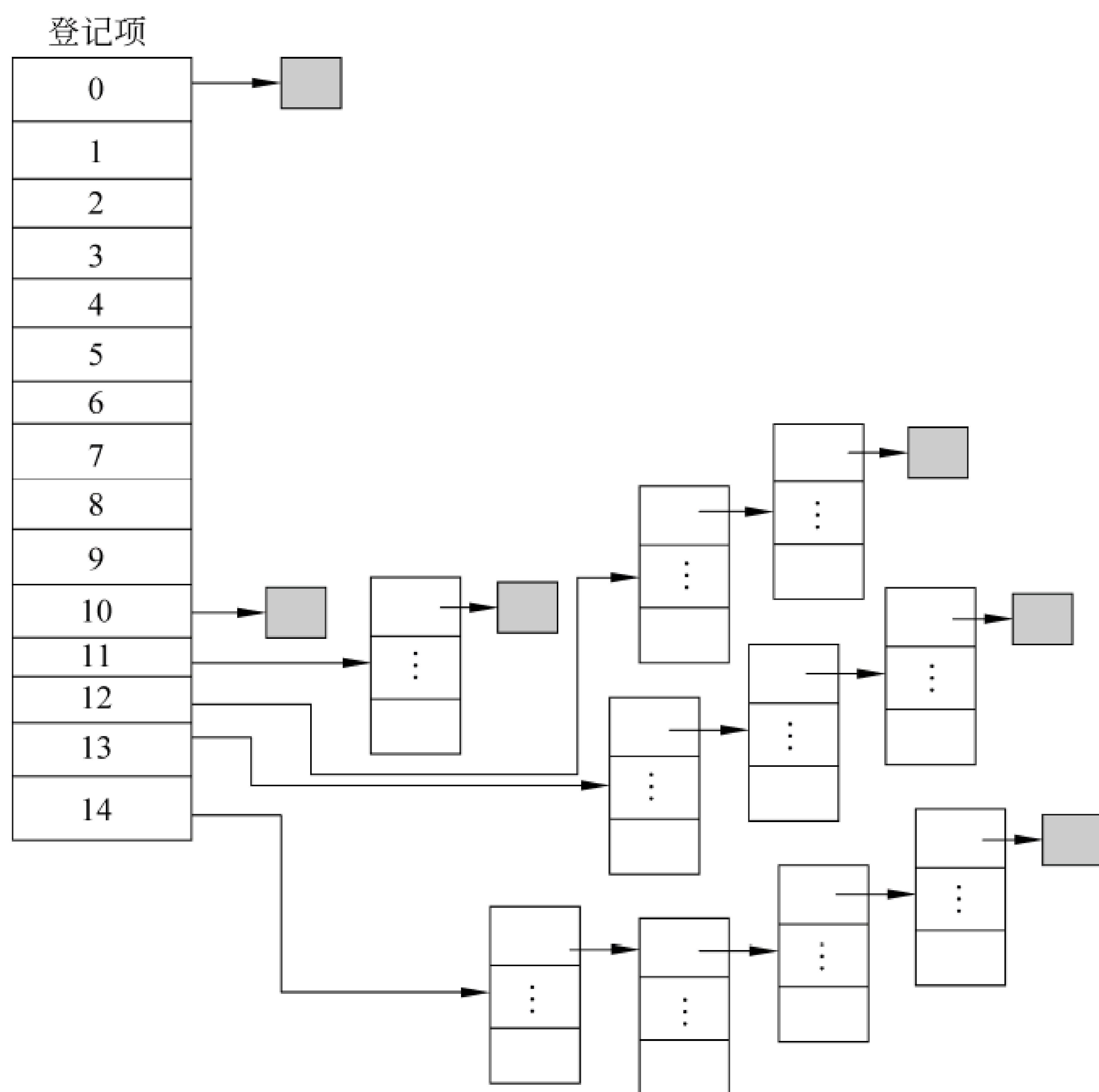


图 6.9　Linux 系统 ext2 文件索引结构

位置如图 6.10 所示。此树形文件目录结构由根目录节点、作为目录文件的中间节点和作为信息文件的叶子节点组成。每个目录项占 127B,每个物理块存放 4 个目录项。根目录的内容常驻内存。请回答下列问题:

(1) 若文件采用链接分配方式,如果要将文件 A 读入内存,至少要存取几次硬盘?为什么?

(2) 若文件采用连续分配方式,如果要将文件 A 的逻辑记录号为 480 的记录读入内存,至少要存取几次硬盘?为什么?

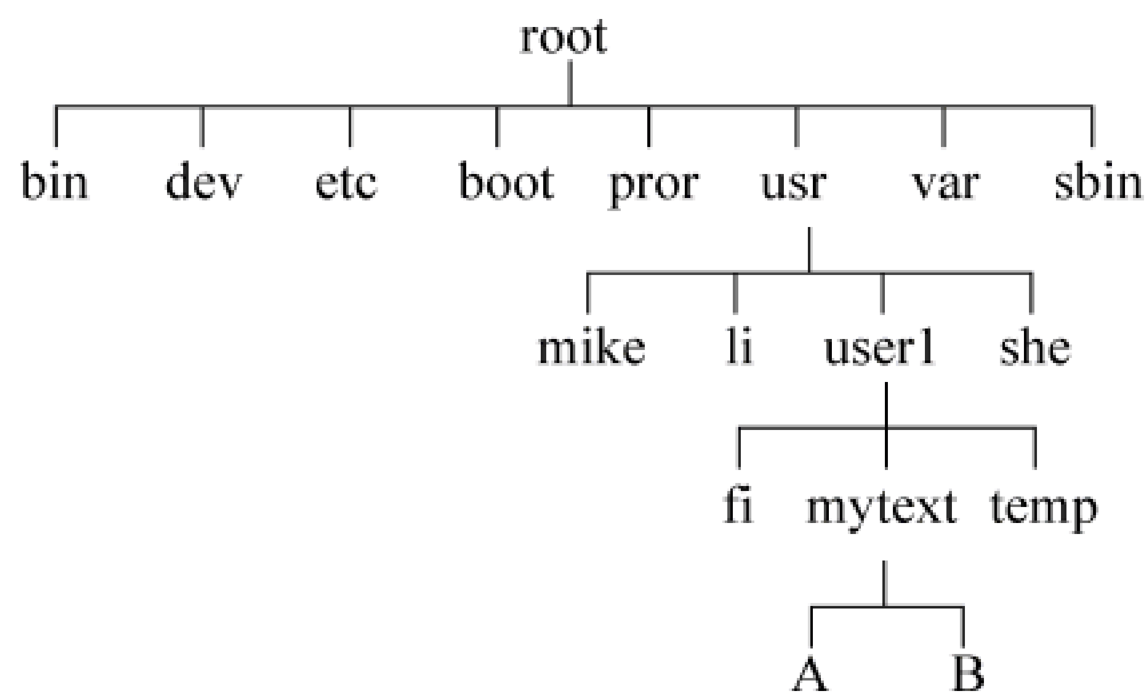


图 6.10　目录结构

解:(1) 首先要检索到文件 A,即通过路径\root\usr\user1\mytext\A 来进行。在最好情况下的步骤如下:

① 从内存中的根目录中找出目录 usr 的目录文件,读入内存(计 1 次硬盘访问)。

② 从目录 usr 的目录文件中找出目录 user1 的目录文件,读入内存(计 1 次硬盘访问)。

③ 从目录 user1 的目录文件中找出目录 mytext 的目录文件,读入内存(计 1 次硬盘访问)。

④ 从目录 mytext 的目录文件中找出文件 A 对应的物理位置,即文件在外存上的存储位置。

文件 A 包含 590 条逻辑记录,需占 590/2=295 个物理块。采用链接分配方式,所有物理块一块一块地读入,因此读入文件 A 需要访问硬盘 295 次。

所以总的存取硬盘次数为 3+295=298 次。

（2）如果采用连续分配方式，同样需要3次硬盘访问得到文件A的起始物理块号S。由于是连续文件，因此可以通过逻辑记录号直接换算成相应的物理块号：$S+480/2=S+240$。要读入该记录只需访问硬盘1次，因此总的访问次数为3+1=4次。

例6.2 有某操作系统对外存分配采用混合索引分配方式，在索引节点中包含文件的物理结构数组iaddr[12]，其中前10项iaddr[0]～iaddr[9]为直接地址，iaddr[10]为一次间接地址，iaddr[11]为二次间接地址。如果系统的盘块大小是4KB，磁盘的每个扇区也为4KB。描述磁盘块的数据项需要4B，其中1B标识磁盘分区，3B标识物理块。请问该文件系统支持的单个文件的最大长度是多少？

解：磁盘块大小为4KB，每个磁盘块要4B标识，则一个磁盘块中可以存放4KB/4=1K个磁盘块号。

采用直接地址的文件长度是10×4KB=40KB。

采用一级间接地址的文件长度是1K×4KB=4MB。

采用二级间接地址的文件长度是1K×lK×4KB=4GB。

该文件系统支持的单个文件的最大长度是40KB+4MB+4GB。

6.3.3 记录的成组与分解

每个用户的文件是由用户按自己的需要组织的。用户对逻辑文件还可按信息在逻辑上的独立含义划分成逻辑记录。显然，逻辑记录的大小是由文件性质决定的。但是，存储介质上的分块与存储介质的特性有关，尤其是磁盘，磁盘上的块是在初始化时预先划好的。因此，逻辑记录的大小往往与存储介质分块的大小不一致。当用户文件的逻辑记录比存储介质的分块小得多时，把一条逻辑记录存入一个块中就会造成存储空间的浪费。为此，可把多条逻辑记录存放在一个块中，当用户需要逻辑记录时再从这一个块的信息中将其分解出来。

1. 记录的成组

把若干条逻辑记录合成一组存入一个物理块的工作称为记录的成组，每块中的逻辑记录条数称为块因子。

记录的成组在不同存储介质上进行信息转储是很有用的，例如，某用户有一批初始数据记录在一叠卡片上，每张卡片上最多记录80个字符。为了方便携带，现要把卡片上的数据转存到磁带上。如果每当卡片输入机读了一张卡片上的信息就立即把它转存到磁带上，则磁带上被划分的块长度也是80个字符。假设磁带的记录密度为一英寸800个字符，而磁带工作时，块与块之间的间隙一般为0.6英寸。于是，当块长为80个字符时，存储信息占用的空间与间隙占用的空间之比为1∶6。显然，磁带空间的利用率极低。若把10张（或更多）卡片的数据集中存放到磁带的一块中，则磁带上的块的长度就增大了，间隙占用的空间比例就减少了，也就提高了磁带空间的利用率。由于信息交换以块为单位，所以，进行成组操作时必须使用主存缓冲区，缓冲区的长度等于最大逻辑记录长度乘以成组的块因子。

有时处理用户作业时，用户要求把产生的中间结果作为文件保存到磁盘上，文件中的逻辑记录是在作业执行过程中陆续形成的。当这些逻辑记录长度较小时，可把它们以成组的方式保存到磁盘上。假设磁盘上的分块长度大于3个逻辑记录的总长，则可先在主存缓冲区中把3个逻辑记录合成一组，然后启动磁盘把3个逻辑记录同时写到磁盘块中。可见，记录的成组不仅提高了存储空间的利用率，而且减少了启动外设的次数，提高了系统的工作

效率。

在实现记录成组时,还应考虑逻辑记录的格式。在记录文件中,每条记录的长度可以是一致的,也可以是不相同的,分别称为定长记录格式和变长记录格式。如果对定长记录格式的文件按记录成组的方式存储到存储介质上,则除最后一块外,每块中存放的逻辑记录条数是相同的。故只要在文件目录中说明逻辑记录的长度和块因子,当需要使用某条记录时仍能方便地将其找出。如果是一个变长记录格式的文件,各个逻辑记录的长度可能不相等,但是每条逻辑记录的长度是确定的,在进行记录成组操作时,应在每条逻辑记录前附加说明记录的控制信息。

2. 记录的分解

从一组逻辑记录中把一条逻辑记录分离出来的操作称为记录的分解。

由于读写存储介质上的信息以块为单位,而用户处理信息要以逻辑记录为单位,所以当逻辑记录成组存储后,用户要处理记录时必须执行记录的分解操作。显然,记录的分解操作也要使用主存缓冲区。

当用户请求读一个文件中的某条记录时,文件系统首先找出该记录所在块的位置,然后把含有该记录的块读入主存缓冲区,再从中分解出指定的记录传送到用户工作区。对定长记录格式,只要按记录的长度就可容易地进行分解。对变长记录格式,要根据附加在记录前说明记录长度的控制信息,计算出用户指定的记录在主存缓冲区中的位置,才能把记录分解出来。由于读入主存缓冲区的一块信息中含有多条逻辑记录,当用户要求读的记录已在主存缓冲区,则可直接从缓冲区中分解出记录传送给用户,否则要启动外设读出一块信息。

从上面的讨论可以看出,记录的成组和分解是以设立主存缓冲区和增加成组分解操作功能为代价的,以此来提高存储介质的利用率和减少启动设备的次数。

例 6.3 某用户文件共 10 条逻辑记录,每条逻辑记录的长度为 480 个字符,现把该文件存放到磁带上,若磁带的记录密度为 800 字符/英寸,块与块之间的间隙为 0.6 英寸,回答下列问题:

(1) 不采用记录成组操作时磁带空间的利用率是多少?

(2) 采用记录成组操作且块因子为 5 时,磁带空间的利用率又是多少?

(3) 当按上述方式把文件存放到磁带上后,用户要求每次读一条逻辑记录存放到他的工作区。当对该记录处理后,又要求把下一条逻辑记录读入他的工作区,直至 10 条逻辑记录处理结束。系统应如何为用户服务?

解:每条逻辑记录占磁带的长度为 480/800=0.6 英寸。

(1) 不采用记录成组操作时,磁带空间的利用率为 0.6/(0.6+0.6)=50%。

(2) 采用记录成组操作且块因子为 5 时,5 条逻辑记录占磁带的长度为 5×0.6=3 英寸,磁带空间的利用率为 3/(5×0.6+0.6)=83.33%。

(3) 当 5 条逻辑记录按成组的方式存放到磁带上后,系统从磁带上读出第一物理块的内容放到指定区域之后,首先分离出第 1 条逻辑记录放到用户的工作区中,待这条逻辑记录处理完后,再分离出第 2 条逻辑记录放到用户的工作区中,待这条逻辑记录处理完后,再分离出下一条记录,直到 5 条逻辑记录处理完;再读下一物理块的内容放到指定区域之后,再做与第一物理块内容相同的处理,直到所有记录处理完。

6.4 文件目录

文件目录是用于组织、检索文件的，是文件系统实现按名存取的重要手段。文件目录由若干目录项组成，每个目录项记录文件的有关信息。在目录项中除了指出文件名和文件在存储介质上的位置外，还应包含如何控制和管理文件的信息。一般情况下，目录项应包含如下内容：

(1) 有关文件存取控制的信息。例如用户名、文件名、文件类型和文件属性(可读写、只读、只可执行等)。

(2) 有关文件结构的信息。例如文件的逻辑结构、文件的物理结构、记录条数和文件在存储介质上的位置等。

(3) 有关文件管理的信息。例如文件的建立日期、文件被修改的日期、文件保留期限和记账信息等。

有了文件目录后，当用户要求使用某个文件时，文件系统查找目录项并比较文件名就可找到指定文件的目录项，根据该目录项中的有关信息可进行核对使用权限等工作，并读出文件供用户使用。因此，文件目录的组织和管理应便于检索和防止冲突。

6.4.1 一级目录结构(单级目录结构)

一级目录结构是最简单的目录结构，所有的文件都登记在同一个文件目录中。图6.11是一级文件目录结构，其中各方框表示文件目录中的各目录项，圆圈表示由目录项描述的文件。

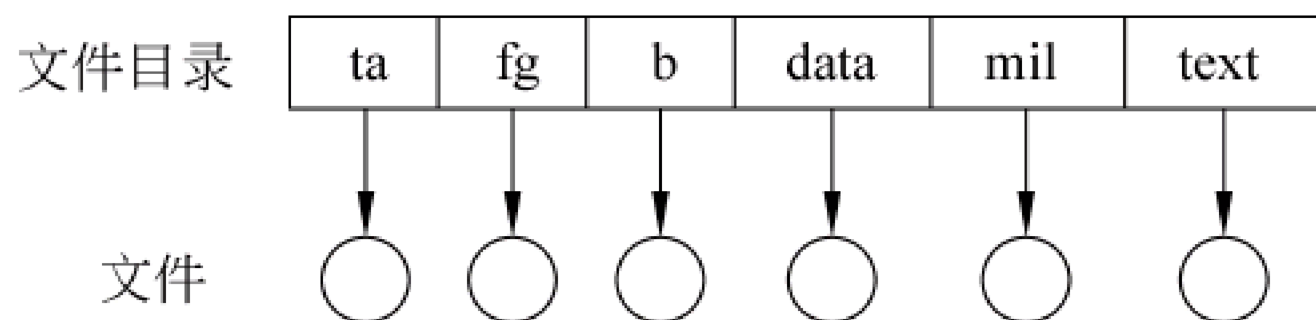


图6.11 一级目录结构

一级目录结构简单，管理方便。每当建立一个新文件时，就在文件目录中增加一个目录项；每当删去一个文件时，就在文件目录中删去该文件的目录项。

一级目录结构要求在文件目录中登记的各个文件都有不同的文件名，如果有重名的话，则在进行“按名存取”时就可能出错。对一个用户来说，当要为一个新文件命名时，必须记住原有文件的文件名，对自己的所有文件都应定义成不同的文件名。但在多道程序设计系统中，若要求所有的用户定义的文件名都不相同是很困难的。所以，一级目录结构一般只适用于微型计算机的单用户系统。

6.4.2 二级目录结构

一级目录结构的主要问题是不允许文件重名。解决重名问题的一种方法是为每个用户建立一个独立的文件目录，于是就形成了二级目录结构。

在二级目录结构中，第一级为主文件目录(Main File Directory，MFD)，它以用户名为索引，对每个用户都设置一个指向用户文件目录(User File Directory，UFD)的指针。第二级为用户文件目录，它为本用户的每一个文件设置一个目录项。图6.12所示是二级目录

结构。

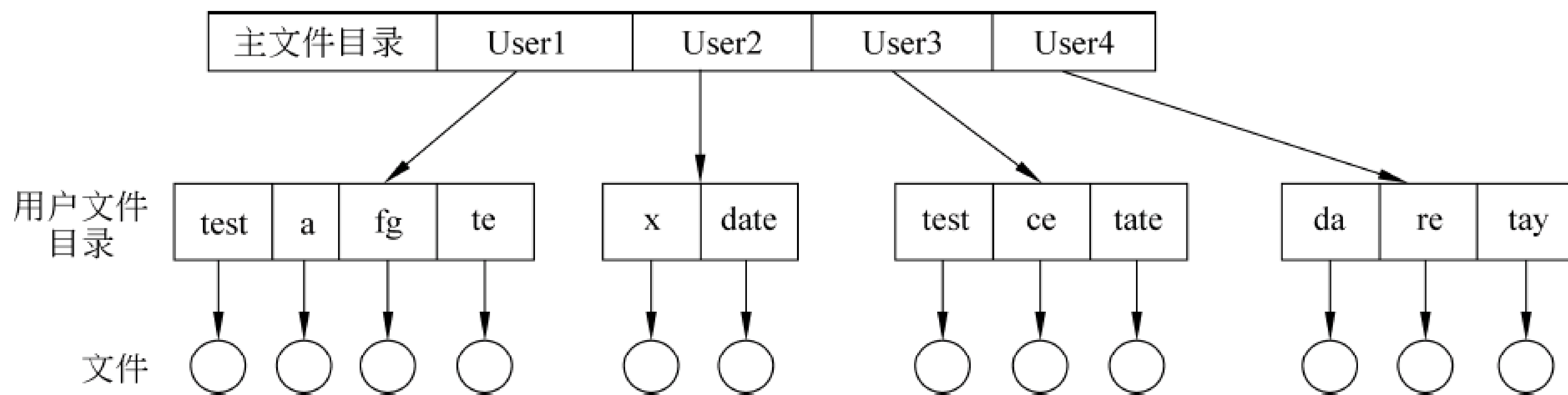


图6.12 二级目录结构

当一个新的用户作业进入系统时,系统应为该用户设置一个用户文件目录,且将其登记到主文件目录中。当用户要访问某个文件时,系统先从主文件目录中找出该用户的用户文件目录,然后在相应的用户文件目录中查找指定的文件。

采用二级目录结构后,用户要求存取文件时,总是搜索该用户自己的文件目录。因此,即使不同的用户在为各自的文件命名时取了相同的文件名也不会引起混乱。但是,在同一个用户文件目录中的各个文件的文件名应该是不相同的。例如,图6.11中的User1和User3都为自己的某个文件取名为test,当User1或User3要查找test文件时,文件系统将分别检索他们各自的文件目录,找到不同的文件存放地址,从而User1或User3总可得到自己所需要的文件信息。同样地,当用户要删除文件时,也不会删除其他用户的同名文件。

如果多个用户要共享某个文件,则只需让各用户文件目录中的相应目录项指向同一个文件的存放地址,这样,各用户就都可以存取该文件,实现了共享。对共享文件,各用户可以定义相同的文件名,或不同的文件名。实际上,在二级目录结构中,文件系统把用户名和文件名联合起来作为各文件的标识。

6.4.3 树形目录结构

如果允许用户在自己的文件目录中根据不同类型的文件再建立子目录,则可把二级目录结构推广成多级目录结构。多级目录结构像一棵倒置的有根树,故把它称为树形目录结构。图6.13描述了树形目录结构。

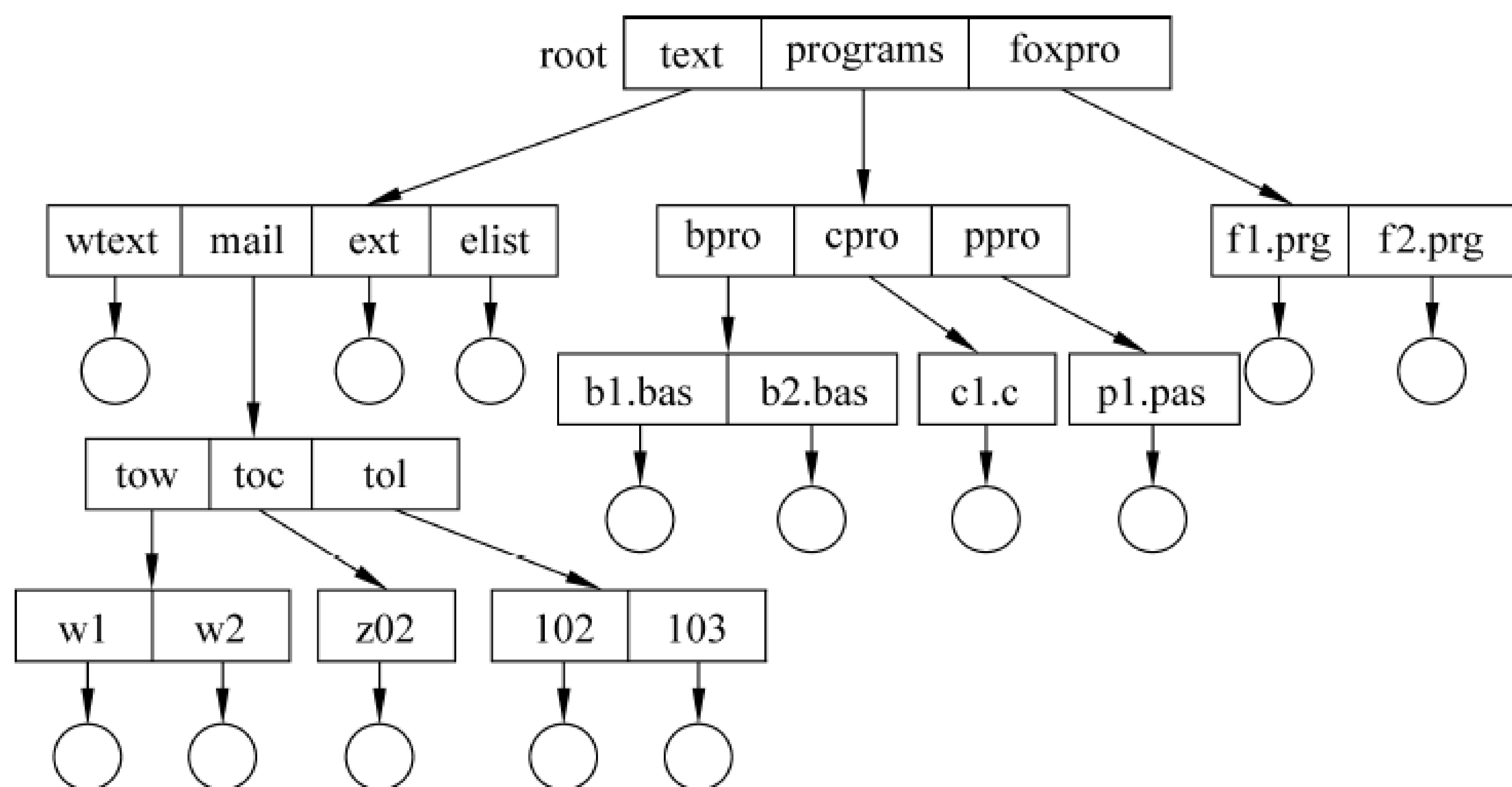


图6.13 树形目录结构

树形目录结构中，主文件目录是树根，因此，常常把主文件目录称为根目录(root)。允许用户在根目录下建立子目录和组织文件。子目录可以是多层次的，就像是树枝，文件在最末端，就像是树叶。在UNIX、MS-DOS、Windows和Linux系统中都采用了树形目录结构。

在树形目录结构中，每个文件都有一个从根到叶的路径。从根目录出发到某文件的通路上所有各级子目录名和该文件名的顺序组合称为文件的路径名，在各级子目录名和文件名之间可用“/”隔开。

每个文件都有一个唯一的路径名，用户存取文件时必须给出文件所对应的路径名。文件系统根据用户指定的路径名检索各级目录，从而确定文件所在的位置。由于查找文件总是从根目录开始，因而查找的时间较长。事实上，用户在一段时间内会经常访问一个子目录下的文件。为了提高效率和方便用户，文件系统引进了“当前目录”的概念。系统初始启动后，当前目录就是根目录。以后，用户可以用文件系统提供的“改变当前目录”命令指定自己当前的工作目录。

有了当前目录后，文件系统把路径名分成两类：绝对路径名和相对路径名。绝对路径名指出了从根目录开始跟随的一条指向指定文件的路径，相对路径名指出了从当前目录出发到指定文件的路径。如果文件就在当前目录中，则存取文件时只要指出文件名就行，文件系统将在当前目录中寻找该文件。如果文件不在当前目录中，但在当前目录的下级目录中，则可用相对路径名指定文件，文件系统就从当前目录开始沿着指定的路径查找该文件。使用相对路径名可以减少查找文件所花费的时间。

在树形目录结构中，根目录或子目录中的目录项可能指向文件，也可能指向下一级子目录。由于每个目录项都有相同的形式，为了区分它指向的是文件还是子目录，可以在目录项中用一位二进制位区分该目录项所指向的是文件(当二进制位取值为“0”时)还是子目录(当二进制位取值为“1”时)。用户可以请求系统为其建立和删除一个文件，或者建立和删除一个子目录。

树形目录结构具有如下优点：

(1) 解决了重名问题。允许在不同的子目录中使用相同的名字命名文件或下级子目录。这样，系统在检索时使用的路径名是不同的，故同名文件不会引起混淆。

(2) 有利于文件的分类。系统或用户可以把不同类型的文件登录在不同的子目录下，并可按层次建立子目录。

(3) 提高检索文件的速度。利用当前目录和相对路径不仅方便用户，而且系统从当前目录开始检索文件，缩短了检索路径，提高了检索速度。

(4) 能进行存取权限的控制。在子目录中可规定存取权限，在检索文件时核对存取权限，避免一个用户未经授权就存取另一个用户的文件，保证了用户文件的私有性，可实现对文件的保护和保密。

6.4.4 文件目录管理

操作系统要管理许多用户的大量文件，因此，系统中也就有许多的文件目录(或子目录)。如果把文件目录都存放在主存储器中，则会占去大量的主存空间。事实上，任何一个用户，在一段时间里只使用少数文件，也就仅涉及少量文件目录。所以，一般系统只把当前正在使用的文件的文件目录存放在主存储器中，称为“值班目录”。

为了对文件目录进行管理，通常把文件目录也作为文件保存在辅助存储器中。由文件目录组成的文件称为目录文件，目录文件可以像其他文件一样进行读/写。文件系统可以根据用户的要求从目录文件中找出用户的当前目录，把当前目录读入主存作为值班目录。这样，既不占用太多的主存空间，又可减少搜索目录的时间。

6.5 磁盘存储空间的管理

磁盘具有大容量的存储空间，它被操作系统和许多用户所共享，用户程序执行期间经常要求在磁盘上存储文件和删除文件，因此，文件系统必须对磁盘空间进行管理。当用户要求存储文件时就要为它分配存储空间，当删除文件时又要收回文件占用的存储空间。常用的磁盘空间管理方法有位示图、空闲块表和空闲块链。

6.5.1 位示图

由于磁盘被划分块后，每一块的大小都是一样的，所以对每个磁盘可以用一张位示图指示磁盘空间的使用情况。一个磁盘的分块确定后，根据总块数决定位示图由多少字组成，位示图中的每一位与一个磁盘块对应，某位为“1”表示相应块已被占用，为“0”表示所对应的块是空闲块。假定有一个盘组共有 100 个柱面（用柱面号标识，编号为 0～99），每个柱面有 8 个磁道（用磁头号标识，编号为 0～7），每个盘面分成 4 个扇区（用扇区号标识，编号为 0～3）。那么，整个磁盘空间共有 4×8×100＝3200 个磁盘块可用来存储信息。如果用字长为 32 位的字来构造位示图，共需 100 个字，如图 6.14 所示。位示图中第 i 个字的第 j 位对应的块号为：块号＝$32i+j$。

	0 位	1 位	2 位	…	30 位	31 位
第 0 字	0/1	0/1	0/1	…	0/1	0/1
第 1 字	0/1	0/1	0/1	…	0/1	0/1
第 2 字	0/1	0/1	0/1	…	0/1	0/1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
第 99 字	0/1	0/1	0/1	…	0/1	0/1

图 6.14 位示图

当有文件要存放到磁盘上时，查位示图中为“0”的位，表示对应的磁盘块空闲可供使用。根据查到的位所在的字号和位号可计算出对应的块号，同时在该位填上占用标志“1”。再由块号计算它所在柱面、盘面和扇区，文件信息就可按确切的地址存放到对应的磁盘块上。

当删除文件归还存储空间时，可以根据归还块的块号或根据归还块所在的柱面号、磁头号和扇区号计算对应位示图中的字号和位号，将所对应的位置为“0”。

6.5.2 空闲块表

系统为每个磁盘建立一张空闲块表，表中每个登记项记录一组连续空闲块的首块号

和块数,其中空闲块数为"0"的登记项为"空"登记项,如图6.15所示。

首块号	空闲块数
19	38
78	12
109	7
⋮	⋮

图6.15 空闲块表

这种管理方式适合采用顺序结构的文件。存储文件时从空闲块表中找一组连续的空闲块进行分配,删除文件时把归还的一组连续块登记到空闲块表中。空闲块的分配和回收算法类似主存的可变分区管理方式。

同样,对分配或回收的块要进行块号与地址之间的换算。

6.5.3 空闲块链

把所有的空闲块连接在一起构成空闲块链,分配空间时从链中取出空闲块,归还空间时把归还块加入到链中。这种管理方式不需要外加专门记录空闲块分配情况的表格。空闲块的连接方式有两种:单块连接和成组连接。

1. 单块连接

把所有空闲块用指针连接起来,每一个空闲块中都设置一个指向另一个空闲块的指针,所有的空闲块就构成了一个空闲块链。系统设置一个链首指针,指向链中的第一个空闲块,最后一个空闲块中的指针为"0"。

分配一块时,根据链首指针把链头的一块分配给申请者,并修改链首指针。归还一块时,把归还块加入到链头,链首指针应指向归还块。利用该方法,每分配或回收一块,都需要启动磁盘进行读/写的工作,这是非常麻烦和费时的。

2. 成组连接

把空闲块分成若干组,把指向一组中各空闲块的指针(即空闲块的块号)集中在一起,这样既可方便查找,又可减少为修改指针而启动磁盘的次数。UNIX系统就是采用空闲块成组连接的方法,图6.16是UNIX系统的空闲块成组连接示意。

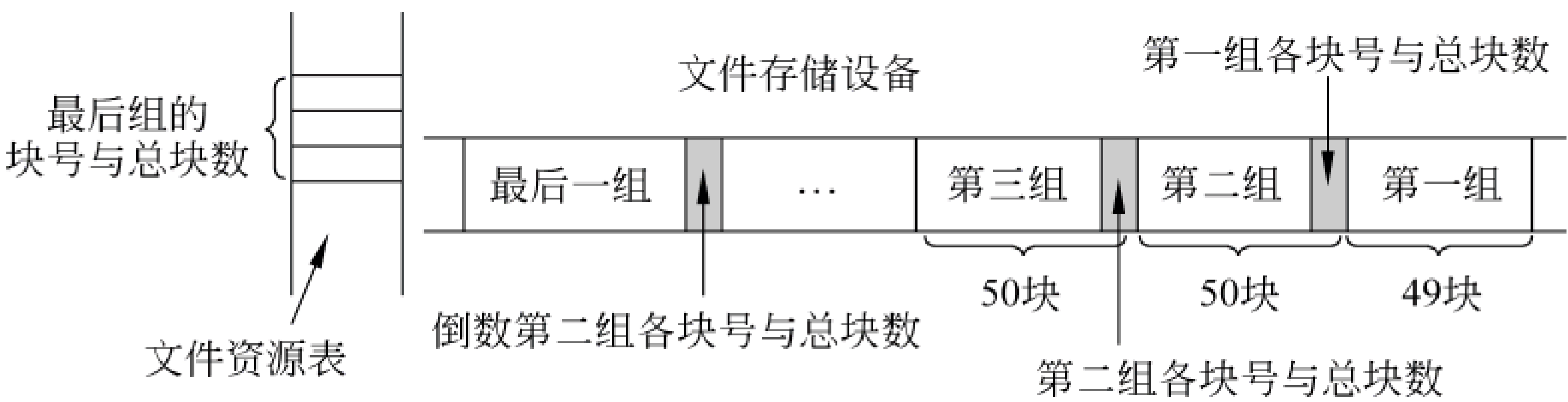


图6.16 UNIX系统的空闲块成组连接示意图

UNIX系统把每50个空闲块作为一组(第一组为49个),每一组的第一个空闲块中登记下一组空闲块的块号和空闲块数,余下不足50块的那部分空闲块的块号及块数登记在一个专用块(在文件资源表中记载)中。

系统初始化时先把专用块内容读到主存。假设初始化时系统已把专用块读入主存中L单元开始的区域中,分配和回收的算法如下。

1) 分配一个空闲块

```
查 L单元内容 (空闲块数);
```

```
if 空闲块数>1
  then  i:=L+空闲块数;
      从 i单元得到一个空闲块号;
      把该块分配给申请者;
      空闲块数减 1。
  else if 空闲块数=1
      then 从 L单元中取得下一组空闲块的块号;
          if 空闲块数=0
              then 申请者等待;
          else 把该块内容复制到专用块;
              把该块分配给申请者;
              把专用块内容读到主存 L。
```

2)归还一块

```
查 L单元的空闲块数:
if 空闲块数<50
  then 空闲块数加 1;
      j:=L+空闲块数;
      归还块号填入 j单元。
if 空闲块数=50
  then 把主存登记的信息写入归还块中;
      把归还块号填入 L+1单元;
      将 L单元置成 1。
```

采用成组连接后,分配回收磁盘块时均在主存中查找和修改,只是在一组空闲块分配完或空闲的磁盘块构成一组时才启动磁盘进行读/写。因此,成组连接的管理方式比单块连接方式效率高。

6.6 磁盘容错技术

容错技术是通过在系统中设置冗余部件来提高系统可靠性的一种技术。磁盘容错技术则是通过增加冗余的磁盘驱动器和磁盘控制器等,来提高磁盘系统的可靠性,即当磁盘系统的某部分出现缺陷或故障时,磁盘仍能正常工作,而且不至于造成数据的错误和丢失。目前,不论是在中、小型机系统还是网络服务器中,都广泛采用磁盘容错技术来改善磁盘系统的可靠性,构成了实际上的稳定存储器系统。磁盘容错技术也称为系统容错技术(System Fault Tolerance,SFT)。它可分为3个级别:SFT-1是低级磁盘容错技术,主要用于防止磁盘表面发生缺陷所引起的数据丢失;SFT-2是中级磁盘容错技术,主要用于防止磁盘驱动器和磁盘控制器故障所引起的系统不能正常工作;SFT-3是高级系统容错技术。

6.6.1 第一级容错技术

第一级容错技术(SFT-1)是最早出现的,也是目前一直在使用的最基本的一种磁盘容错技术。它包含双份目录、双份文件分配表及写后读校验等措施。

1. 双份目录和双份文件分配表

在磁盘上存放的文件目录和文件分配表(FAT)是文件管理所用的重要数据结构。它记录了文件的属性、文件在磁盘上的物理地址等重要数据。如果这些表格被破坏,将导致磁盘上的部分或全部文件成为不可访问的,也就等于文件丢失。为了防止这类情况发生,可在不同的磁盘上或在磁盘的不同区域中分别建立两份目录表和 FAT。一份称为主目录及主 FAT,另一份则称为备份目录及备份 FAT。一旦由于磁盘表面缺陷而造成文件目录或 FAT 损坏时,系统便自动启用备份文件目录和备份 FAT。从而可以保证磁盘上的数据仍是可访问的,并将损坏区写入坏块表中,系统还要在磁盘的其他区域再建立新的文件目录或 FAT 作为备份。在系统每次加电启动时,都要对两份目录和两份 FAT 进行检查,以验证它们的一致性。

2. 热修复重定向和写后读校验

通常只有在磁盘上有较多缺陷或完全损坏时,才换一个新盘,而对于磁盘表面有少量缺陷的情况,则多是采取其他补救措施后继续使用。补救措施主要用于防止将数据写入有缺陷的盘块中。

1) 热修复重定向(hot-fix redirection)方式

系统将一定的磁盘容量(例如 2%~3%)作为热修复重定向区,用于存放当发现盘块有缺陷时的写入数据,并对写入该区的所有数据进行登记,以便以后对数据进行访问。

2) 写后读校验(read after write verification)方式

为了保证所有写入磁盘的数据都能写入到完好的盘块中,应该在每次从内存缓冲区向磁盘中写入一个数据块后,又立即从磁盘上读出该数据块,送至另一缓冲区中,再将该缓冲区中内容与内存缓冲区中在写后仍保留的数据进行比较,若两者一致,便认为此次写入成功,可继续写下一个盘块,否则,再重写。若重写后两者仍不一致,则认为该盘块有缺陷,此时,便将应写入该盘块的数据写入热修复重定向区中,并将该损坏盘块的地址记录在坏盘块表中。

6.6.2 第二级容错技术

1. 磁盘镜像(disk mirroring)

SFT-1 只能用于防止由于磁盘表面部分故障造成的数据丢失,但如果磁盘驱动器发生故障,用 SFT-1 级容错便无能为力,仍可能造成数据丢失。为了避免在这种情况下的数据丢失,便增设了磁盘镜像功能。为实现该功能,需在同一磁盘控制下再增设一个完全相同的磁盘驱动器,如图 6.17 所示。



图 6.17 磁盘镜像示意图

采用磁盘镜像工作方式时,在每次向文件服务器的主磁盘写入数据后,都要采用写后读校验方式,将数据再同样地写到备份磁盘上。使两个磁盘上有着完全相同的位像图。或者说,可把备份磁盘看做是主磁盘的一面镜子。当其中一个磁盘驱动器发生故障时,由于备份磁盘的存在,在进行切换后,文件服务器仍能正常工作,从而不会造成数据的丢失,这便是第二级容错技术(SFT-2)。在一个磁盘驱动器发生故障时,必须立即发警告,尽快修复,以恢复磁盘镜像功能。磁盘镜像虽然实现了容错功能,但并未能使服务器的

磁盘 I/O 速度得到提高,而且磁盘的利用率仅为 50%。

2. 磁盘双工(disk dupluxing)

磁盘镜像功能虽能有效地解决在一台磁盘机故障时的数据保护问题,但如果控制这两台磁盘驱动器的磁盘控制器或主机到磁盘控制器之间的通道发生了故障,则将使这两台磁盘机同时失效,起不到数据保护作用。如此,在第二级容错技术中,又增加了磁盘双工功能,以在磁盘控制器发生故障时起到数据保护作用。所谓磁盘双工,是指将两台磁盘驱动器分别接到两个磁盘控制器上,同样地使这两台磁盘机镜像成对,如图 6.18 所示。

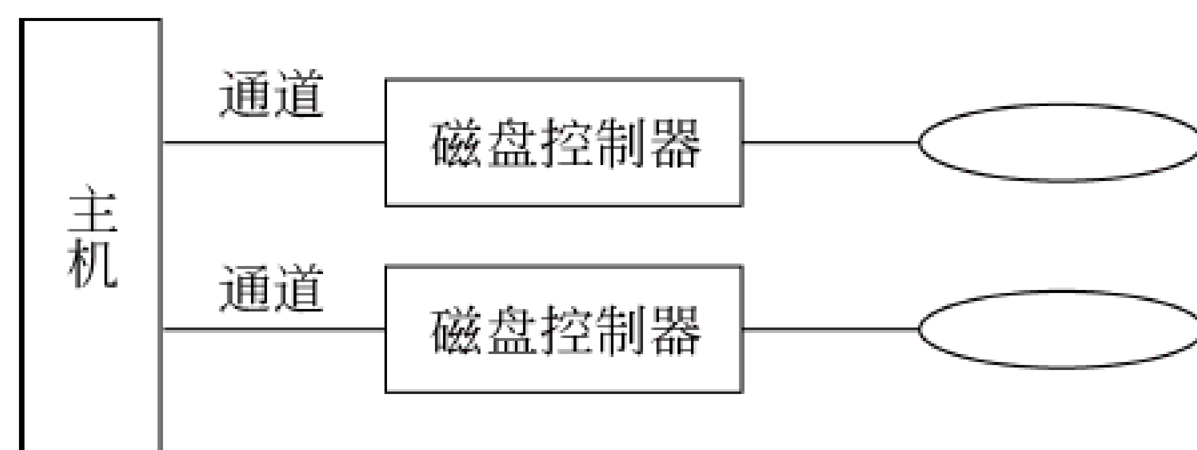


图 6.18 磁盘双工示意图

在磁盘双工时,文件服务器同时将数据写到两个处于不同控制器下的磁盘上,使两者有着完全相同的位像图。如果某个通道或控制器发生故障时,另一通道上的磁盘仍能正常工作,这样便不会造成数据的丢失,同时须立即发出警告。以便尽早恢复磁盘双工功能。在磁盘双工时,由于每一个磁盘都有着自己的独立通道,故可同时(并行)地将数据写入磁盘。在读数据时,可采取分离搜索(split seek)技术,从响应快的通道上取得数据,因而加快了对数据的读取速度。

磁盘镜像和磁盘双工在当前计算机系统工程中经常使用,是行之有效的数据保护手段。值得注意的是磁盘镜像和磁盘双工技术都比较复杂,且在磁盘中都可能保存许多有用的数据,误操作很容易造成数据的丢失,甚至造成硬件的损坏,因此,这两种功能的操作最好交由比较熟练的专业人员去负责。

6.6.3 廉价磁盘冗余阵列

廉价磁盘冗余阵列(Redundant Arrays of Inexpensive Disk,RAID)是在 1987 年由美国加利福尼亚大学伯克利分校提出的,现在已开始广泛地应用于大、中型计算机系统和计算机网络中。它是利用一台磁盘阵列控制器来统一管理和控制一组(几台到几十台)磁盘驱动器,组成一个高度可靠的、快速的大容量磁盘系统。

1. 并行交叉存取

为了提高磁盘的访问速度,把在大、中型机中应用的交叉存取(interleave)技术应用到磁盘存储系统中。在该系统中,有多台磁盘驱动器,系统将每一盘块中的数据分为若干个盘块数据,再把每一个子盘块的数据分别存储到各个不同磁盘中的相同位置上。在以后当要将一个盘块中的数据传送到内存时,采取并行传输方式,将各个盘块中的子盘块数据同时向内存中传输,从而使传输时间大大减少。例如,在存放一个文件时,可将该文件中的第一个数据子块放在第一个磁盘驱动器上,将文件的第二个数据子块放在第二个磁盘上,……,将第 N 个数据子块放在第 N 个驱动器上,以后在读取数据时,采取并行读取方式,即同时从第 $1\sim N$ 个磁盘上将第 $1\sim N$ 个数据子块读出,这样便把磁盘 I/O 的速度提高了 $N-1$ 倍。

2. RAID 的分级

RAID 在刚被推出时分成 6 级,即 RAID 0 级~RAID 5 级,后来又增加了 RAID 6 级和 RAID 7 级。

1）RAID 0 级

该级仅提供了并行交叉存取。它虽能有效地提高磁盘 I/O 速度，但并无冗余校验功能，致使磁盘系统的可靠性不好。只要阵列中有一个磁盘损坏，便会造成不可弥补的数据丢失。故较少使用。

2）RAID 1 级

它具有磁盘镜像功能。可利用并行读/写特性，将数据分块并同时写入主盘和镜像盘，故比传统的镜像盘速度快。但它的磁盘容量的利用率只有 50%，它以牺牲磁盘容量为代价来提高磁盘系统的可靠性。

3）RAID 2 级

这是一种数据以海明编码方式分布于各个磁盘中的磁盘阵列，故称为海明编码阵列。由于其由于开销大而不常用。

4）RAID 3 级

这是具有并行传输功能的磁盘阵列。它利用一台奇偶校验盘来完成容错功能，比起磁盘镜像，它减少了所需要的冗余磁盘数。例如，当阵列中只有 7 个盘时，可用 6 个作数据盘，一个作校验盘，磁盘的利用率为 6/7。RAID 3 级常用于科学计算和图像处理。

5）RAID 4 级

这是一种具有专盘奇偶校验独立存取磁盘阵列。数据以块交叉的方式存于各个盘中，块的大小可变。奇偶校验信息存储在一台专用的磁盘上。

6）RAID 5 级

这是一种具有独立传送功能的磁盘阵列，每个驱动器都有各自独立的数据通路，独立地进行读/写，且无专门的校验盘，用来进行纠错的校验信息是以螺旋(spiral)方式散布在所有数据盘上。RAID 5 级常用于 I/O 较频繁的事务处理。

7）RAID 6 级和 RAID 7 级

这是强化了的 RAID。在 RAID 6 级的阵列中设置了一个专用的、可快速访问的异步校验盘。该盘具有独立的数据访问通路，具有比 RAID 3 级及 RAID 5 级更好的性能，但其性能改进有限，且价格昂贵。RAID 7 级是对 RAID 6 级的改进，在该阵列中的所有磁盘都具有较高的传输速率，有着优异的性能，是目前最高档次的磁盘阵列，但其价格较高。

3. RAID 的优点

RAID 自 1988 年面世后，便引起了人们的普遍关注，并很快流行起来。这主要是因为 RAID 具有下述几个明显的优点：

(1) 可靠性高。RAID 最大的特点就是它的高可靠性。除了 RAID 0 级外，其余各级都采用了容错技术。

(2) 磁盘 I/O 速度高。由于磁盘阵列可采取并行交叉存取方式，故可将磁盘 I/O 速度提高 $N-1$ 倍，其中 N 为磁盘数目。

(3) 性能/价格比高。利用 RAID 技术来实现大容量高速存储器时，与具有相同容量和速度的大型磁盘系统相比，其体积只是后者的 1/3，价格也是后者的 1/3，且可靠性更高。

6.6.4 后备系统

磁盘系统的容量通常都很大，仍不可能将所有的信息都装入其中。在系统运行一段时

间后,就可能将磁盘装满,因此需要每隔一定的时间,就将磁盘上的大部分数据转储到后备系统中。而在后备系统中的数据也需每隔一定的时间重新进行复制,以防止由于自然因素使后备系统中的数据逐渐消失。可见,作为一个完整的应用系统,必须配置后备系统。

1. 后备系统的类型

目前常用的后备系统有3类:磁带机、磁盘机和光盘机。它们各有优缺点。

1) 磁带机

磁带机是最早作为计算机系统的外存设备的,但由于它只适合于存储顺序文件,故现在主要是把它作为后备设备。目前比较流行的两种磁带机是1/4英寸磁带机(Quarter Inch Cartridge,QIC)和数字音频磁带机(Digital Audio Tape,DAT)。由于磁带机有容量大、价格便宜等优点,故几乎在所有的应用系统中都配置了磁带机。其缺点是速度较慢。

2) 磁盘机

当采用磁盘机来做后备系统时,主要可采用两种方式:

(1) 利用活动磁盘机来做后备系统。这种方法的最大优点是速度快,而且其脱机保存期也可比磁带机长出3~5年。但其单位容量的存储费用较高。

(2) 利用大容量磁盘机兼作后备系统。此时需为一个系统配置两个大容量硬盘系统,每个硬盘都被划分成两个区,数据区和备份区。如图6.19所示,每天晚上都要将硬盘驱动器0中的“数据0”复制到磁盘机驱动器1的副本区中;也将磁盘机驱动器1中的“数据1”复制到磁盘机驱动器0的副本区中。采用这种方法时,不仅其复制速度快,而且也具有容错功能,即当其中任一驱动器出现故障时,都不会造成数据的丢失。

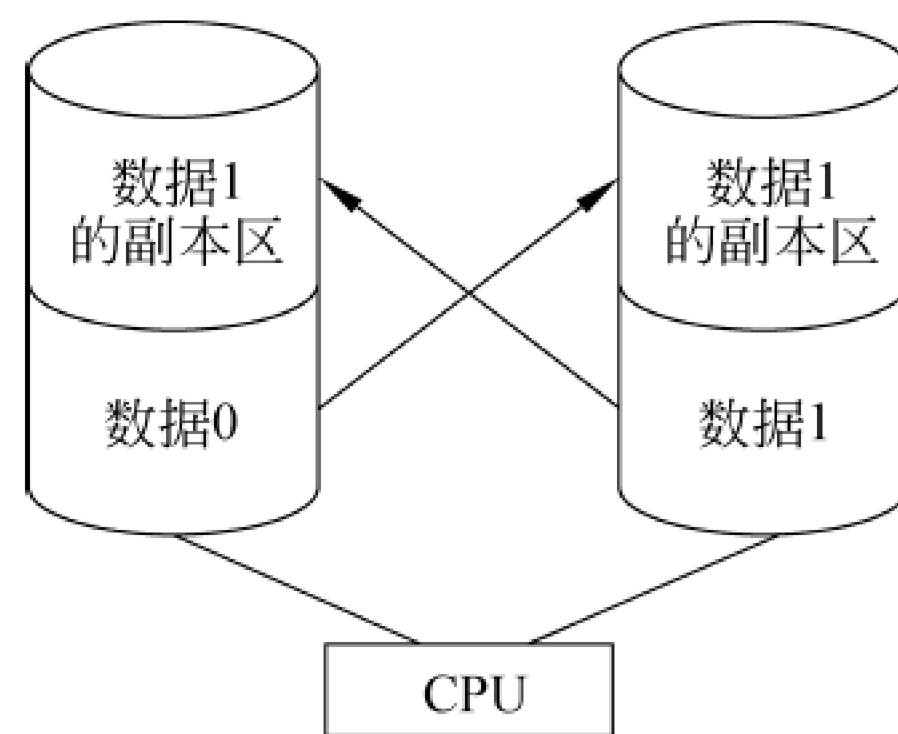


图6.19 利用两个硬盘互为后备系统的示意图

3) 光盘

光盘的种类很多,目前最流行的是CD-ROM(Compact Disk Read-Only Memory)。可用为后备设备的主要有WORM(Write-Once,Read-Many)光盘,它可以写一次,多次读。该盘的典型容量为650MB。另一种是可擦式光盘(erasable optical disk),写于光盘上的数据很容易擦去再进行重写,其典型容量也是650MB。

光盘的容量大,且保存期也特别长,一般可达15年以上,有的甚至可达100年。

2. 复制方法

为将磁盘上的数据复制到后备设备上,可采用下述两种方法。

1) 完全转储法

这是指定期地将磁盘上的整个文件系统复制到后备系统上。此方式简单,但效率低。因为在两次相邻的复制之间,整个文件系统中可能只有一小部分发生了变化,还有很大一部分并无任何变化。因而会产生很多的重复复制。

2) 增量转储(incremental dumps)法

通常,可以把一个月(31或30天)作为一个周期,在一个周期的第一天时,把磁盘上所有的文件复制到后备系统上,第二天只复制在第一天发生了变化的那些文件,而第三天则复制在第二天后发生了变化的那些文件,……,这样,直至月末。到下个月的第一天开始,又重复上述过程。

为了实现增量转储，在系统中应配置一张转储时间表，在其中记录下每个文件最后一次的转储时间。转储程序在进行转储时检查每个文件在最近一次转储后是否又发生了变化。如果这期间尚未发生变化，此次便无须对它进行转储；如果已经修改过时，则需要进行转储，并修改转储时间表中该文件的最后转储时间。这种方法需要占用较多的后备存储介质和机器时间，但由于每天都做了备份，故当有些文件被误删除时，可以进行恢复。

6.7 文件的使用

文件的物理结构对用户来说是不必关心的，但对文件系统来说却是至关重要的，因为它直接影响存储空间的使用和检索文件信息的速度。用户按逻辑结构使用文件，文件系统按物理结构管理文件。因此，当用户请求读写文件时，文件系统必须实现文件的逻辑结构与物理结构之间的转换。

文件系统把用户组织的逻辑文件按一定的方式转换成物理文件存放到存储介质上，当用户需要文件时，文件系统又从存储介质上读出文件并把它转换成逻辑结构。文件系统实现按名存取为用户提供方便，同时也要求用户遵照系统规定和提供的手段使用文件。

6.7.1 文件的操作

为了正确地实现文件的存取，文件系统设计了一组与存取文件有关的功能模块，用户可以用访管指令（陷阱）调用这些功能模块，以实现对文件的存取要求。把文件系统设计的这一组功能模块称为“文件操作”，文件操作主要有以下6种。

1. “建立文件”操作

用户要求把一个新文件存放到存储介质上之前，首先调用文件系统的“建立文件”操作。在请求调用该操作时，用户必须给出如下参数：用户名、文件名、存取方式、存储设备类型、记录格式和记录长度等。

“建立文件”操作的主要工作是检查文件目录，确认无重名时寻找空登记项进行登记，寻找空闲存储块（至少一块）以备存储文件信息或存放索引表。

2. “打开文件”操作

用户要求使用一个已经存放在存储介质上的文件前，首先调用文件系统的“打开文件”操作。用户在请求调用时应给出如下参数：用户名、文件名、存取方式和存储设备类型等。

“打开文件”操作的主要工作是找出用户的文件目录并读入主存；检索文件目录找出与文件名相符的登记项；核对存取方式是否一致；对索引结构的文件还要把该文件的索引表读到主存等。

3. “读文件”操作

用户要求读文件信息时调用文件系统的“读文件”操作。用户在请求调用时应给出的参数是用户名、文件名、存取方式和存放信息时的主存地址等。对随机存取方式的文件，还要说明读哪个记录。

“读文件”操作的主要工作是确认该文件是否已打开且核对是否是打开者请求读文件；核对无误后，对顺序存取方式的文件，每次按逻辑顺序读一个或几个逻辑记录传送到用户指定的主存地址。对随机存取方式的文件，按用户指定的记录号（或键）查索引表，得到该记录

的存储地址后，按地址将记录读出，并传送到用户指定的主存地址。

允许用户对一个已经打开的文件分多次读。

4. “写文件”操作

用户要求存文件信息时，调用文件系统的“写文件”操作。用户调用“写文件”操作时也要给出参数，参数形式同“读文件”操作。

“写文件”操作的主要工作是查找文件目录核对文件是否已建立，若已建立，对顺序存取方式的文件，找出存放文件信息的位置且写入文件信息，同时保留一个“写指针”指出下一次写文件时的存放位置；对随机存取方式的文件，把索引表读入主存，在索引表中找一个空登记项且找一个空闲的磁盘块，把记录存入磁盘块，同时把记录号和记录存放地址填入索引表。允许用户对一个文件分多次写。若文件不存在，则先建立文件，再进行写文件操作。

5. “关闭文件”操作

用户对文件读写完毕后，需要调用文件系统的“关闭文件”操作。用户在请求调用时需给出用户名、文件名和设备类型等参数。

“关闭文件”操作的主要工作是核实只有文件的建立者或打开者才有权关闭文件；检查读入主存的文件目录或索引表是否被修改过，若被修改过，则应把修改过的文件目录或索引表重新写回到存储介质上。

一个被关闭后的文件不能再使用。若要使用则必须再次调用“打开文件”操作。

6. “删除文件”操作

用户认为自己的文件没有必要再保存时，可以调用文件系统的“删除文件”操作。调用时应给出的参数是文件名和设备类型。

“删除文件”操作的主要工作是把用户指定的文件在文件目录中除名并收回文件所占用的存储空间。

6.7.2 文件的使用

从上面介绍的文件操作的功能可以看出，“打开文件”、“建立文件”和“关闭文件”是文件系统中的特殊操作。“打开文件”和“建立文件”两个操作实际起着用户申请对文件使用权的作用，经文件系统验证符合使用权限时才允许用户使用文件，并适当地为用户做好使用文件前的准备工作。“关闭文件”操作的作用是让用户向系统归还文件的使用权。

文件系统提供按名存取功能后，为保证对文件的正确管理和文件信息的安全可靠，规定了用户使用文件的操作步骤。

1. 读文件

用户请求读文件信息时依次调用“打开文件”、“读文件”和“关闭文件”操作。

这样可保证一个文件被打开后，在它被关闭之前不允许非打开者使用，只有打开文件者才有权去读文件，以避免一个共享文件（多个用户都可使用的文件）被多个用户同时使用而造成的混乱。

2. 写文件

用户请求写文件信息时依次调用“建立文件”、“写文件”和“关闭文件”操作。

这样可保证在同一级目录中不会有重名文件，一个文件可分多次写，用“关闭文件”表示有关该文件的信息已经结束。

3. 删除文件

用户请求删除文件时依次调用“关闭文件”和“删除文件”操作。

一个正在使用的文件是不允许删除的，所以，只有先归还文件的使用权后才能删除文件。

有的系统为了方便用户，提供一种隐式使用文件的方法，允许用户不调用“打开文件”和“建立文件”的操作，而直接调用“读文件”或“写文件”操作。当用户要求使用一个未被打开或未被建立的文件时，文件系统先做“打开文件”或“建立文件”操作，然后再执行“读文件”或“写文件”操作。当用户使用了一个A文件后又要使用B文件，文件系统就先关闭A文件，再打开或建立B文件，然后对B文件执行读/写操作。

提供隐式使用的系统中，用户既可显式地提出“打开文件”、“建立文件”和“关闭文件”的要求，也可不直接提出这些要求。但是，在用户采用隐式使用时，文件系统仍必须做这些工作。

6.8 文件的共享、保护和保密

6.8.1 文件的共享

文件共享是指一个文件可以让指定的多个用户共同使用。文件共享有许多好处，例如，免除系统复制文件的工作，节省文件占用的存储空间等。

在允许文件共享的系统中，必须对共享文件进行管理，共享文件的使用有如下两种情况：

(1) 不允许同时使用。任何时刻只允许一个用户使用共享文件，即不允许两个或两个以上的用户同时打开一个文件。一个用户打开共享文件后，待使用结束关闭文件后，才允许另一个用户打开该文件。

(2) 可以同时使用。允许多个用户同时使用同一个共享文件，但系统必须实现对共享文件的同步控制。一般说，允许多个用户同时打开共享文件执行读操作，而不允许读/写者同时使用共享文件，也不允许多个写同时对共享文件执行写操作，以确保文件信息的完整性。相关方面的内容可以参阅第8章的经典进程同步问题。

文件系统的一个重要任务就是为用户提供共享文件信息的手段。

实现文件共享的主要方法有：

(1) 绕道法；

(2) 链接法；

(3) 基本文件目录表(Base File Directory，BFD)。

绕道法要求每个用户处在当前目录下工作，用户对所有文件的访问都是相对于当前目录进行的。用户文件的固有名(为了访问某个文件而必须访问的各个目录的目录名与文件名的顺序连接称为固有名)由当前目录到信息文件通路上所有各级目录的目录名加上该信息文件的符号名组成。使用绕道法进行文件共享时，用户从当前目录出发向上返回到与所要共享的文件所在路径的交叉点，再顺序向下访问到共享文件。绕道法需要用户指定所要共享文件的逻辑位置或到达被共享文件的路径。绕道法的原理如图6.20所示。

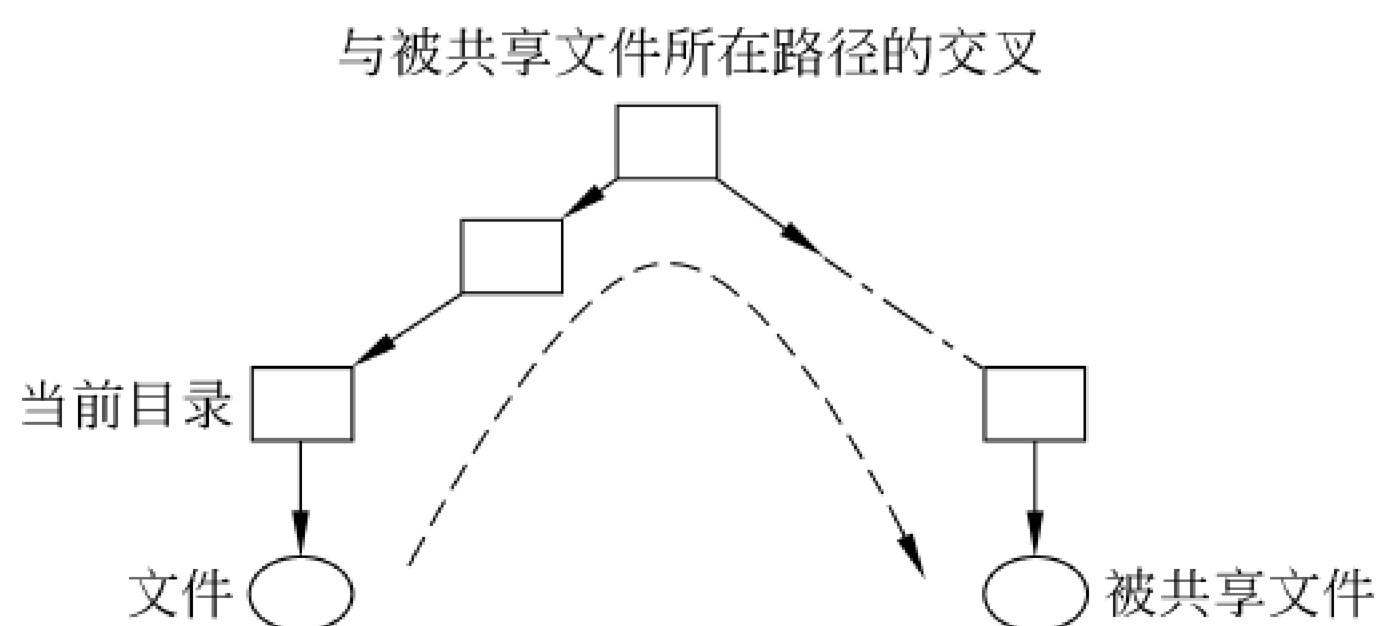


图 6.20　绕道法

显然,绕道法要绕弯路访问多级目录,因而其搜索效率不高。为了提高共享其他目录中文件的速度,另一种共享的办法是在相应目录表之间进行链接,即将一个目录中的链指针直接指向被共享文件所在的目录。链接法仍然需要用户指定被共享的文件和被链接的目录。

实现文件共享的一种有效方法是采用基本文件目录表的方法。该方法把所有文件目录的内容分成两部分:一部分称为符号文件目录表(System File Directory,SFD)包括文件的结构信息、物理块号、存取控制和管理信息等,并由系统赋予唯一的内部标识符来标识;另一部分是基本文件目录表(BFD),由用户给出的符号名和系统赋予的文件说明信息的内部标识符组成。SFD 中存放文件名和文件内部标识符,BFD 中存放除了文件名之外的文件说明信息和文件的内部标识符。这样组成的多级目录结构如图 6.21 所示。

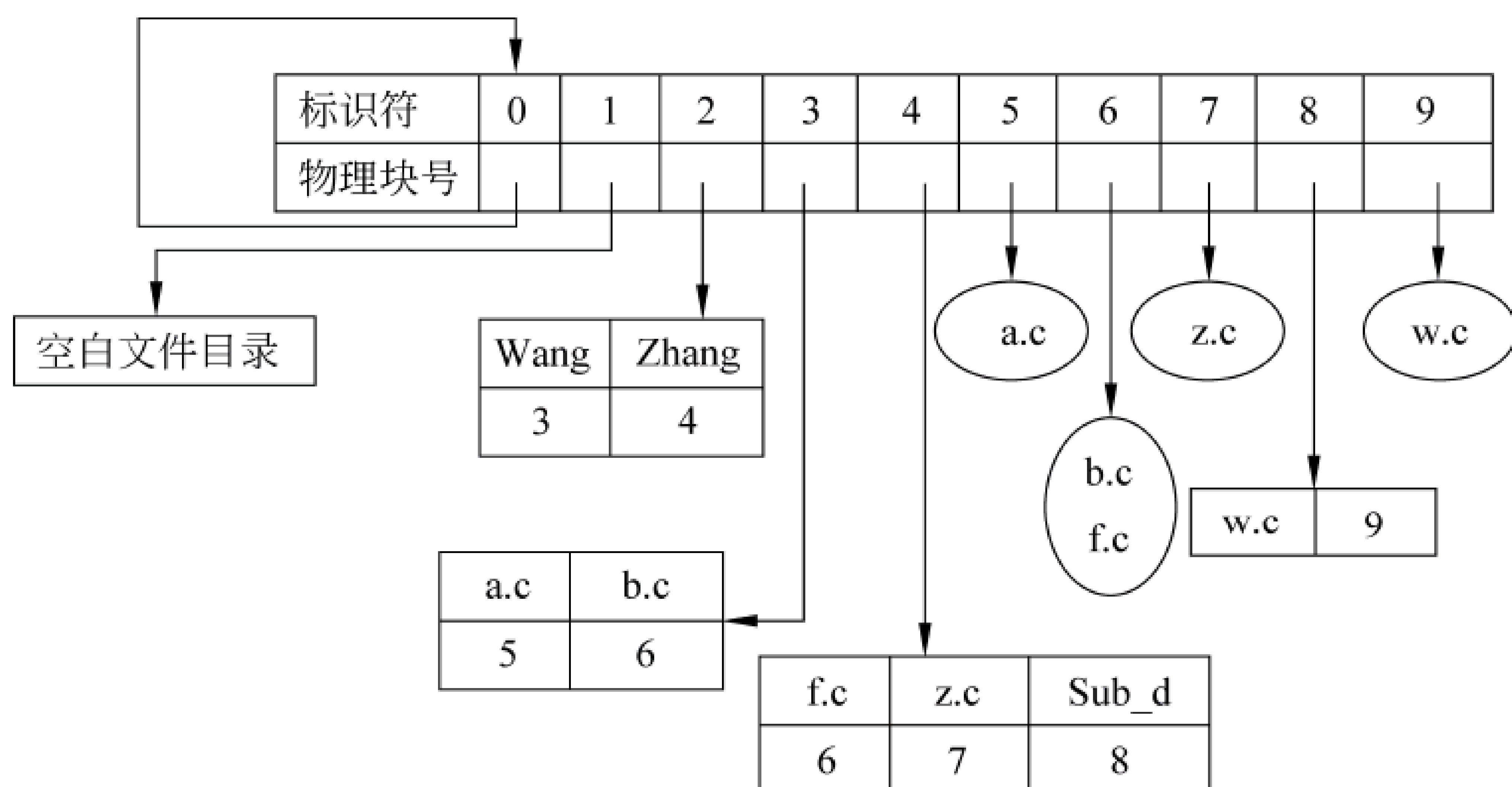


图 6.21　采用基本文件目录的多级目录结构

在图 6.21 中,为了简单起见,未在 BFD 表项中列出结构信息、存取控制信息和管理控制信息等。另外,在文件系统中,系统通常预先规定赋予基本文件目录、空白文件目录、主目录 MFD 的符号文件目录固定不变的唯一标识符,在图中它们分别为 0、1、2。

采用基本文件目录方式可较方便地实现文件共享。如果用户要共享某个文件,则只需给出被共享的文件名,系统就会自动在 BDF 的有关文件处生成与被共享文件相同的内部标识符 ID,例如在图 6.21 中,用户 Wang 和 Zhang 共享标识符为 6 的文件,对于系统来说,标识符 6 指向同一个文件,而对 Wang 和 Zhang 两个用户来说则对应于不同的文件名 b.c 和 f.c。

6.8.2 文件的保护

文件的保护是防止文件被破坏。造成文件可能被破坏的原因，有时是硬件故障或软件失误引起的，有时是由于用户共享文件时发生错误引起的，文件系统应根据不同的情况采用不同的保护措施。

1. 防止系统故障造成的破坏

文件系统必须有防止硬、软件的各种意外可能破坏文件的能力。为此，文件系统经常采用建立副本和定时转储的方法来保护文件。

1）建立副本

把同一个文件保存到多个存储介质上，这些存储介质可以是同类的，也可以是不同类型的。这样，当对某个存储介质保管不善而造成文件信息丢失时，或当某类存储设备故障暂不能读出文件时，就可用其他存储介质上的备用副本来替换。这种方法简单，但设备费用和系统开销增大，当文件需修改或更新时，必须要改动所有的副本。因此，这种方法一般用于短小且极为重要的文件。

2）定时转储

每隔一定的时间把文件转储到其他的存储介质上，当文件发生故障时，就用转储的文件来复原，把有故障的文件恢复到某一时刻的状态。这样，仅丢失了自上次转储以来新修改或增加的信息，从恢复后的状态开始重新执行。

2. 防止用户共享文件可能造成的破坏

对共享文件要防止非法使用文件造成的破坏，这就涉及用户对文件的使用权限。对文件的使用权限可以分成只准读、可读可写、只准执行和有权删除等。可以用下面的方法规定用户使用文件的权限：

1）采用树形目录结构

凡能得到某级目录的用户就可得到该级目录所属的全部目录和文件，按目录中规定的存取权限使用目录或文件。

2）存取控制矩阵

列出每个用户对每个文件或子目录的存取权限，可以用一个二维矩阵表示，如图 6.22 所示。

文件标识 / 用户标识	1	2	…	m
1	R	RWX	…	RX
⋮	⋮	⋮	⋱	X
n	D	W	…	RXW

（R：读，W：写，X：执行，D：删除）

图 6.22　存取控制矩阵

矩阵中的每一个元素 a_{ij} 规定了第 i 个用户对第 j 个文件的存取权限。当某个用户对一个文件提出使用要求（读、写、执行或删除）时，系统按存取权限的规定与用户的使用要求比较，当相符合时才允许使用这个文件。

存取控制矩阵的最大问题是用户与文件较多时这个矩阵就很大,实现起来系统开销很大。为此引入存取控制列表和权能列表(具体在第9章中介绍)。

3) 文件使用权限

在许多系统中把与每个文件相关的用户分成3类:文件主、伙伴和一般用户。文件主是文件的建立者,伙伴是指可共享文件主建立的文件且具有相同存取方式的一组用户,一般用户是指除文件主及其伙伴外系统中的所有其他用户。对每一类用户规定使用文件的权限,当用户提出使用某个文件的要求时,系统先检查该用户是文件主还是文件主的伙伴或是一般用户,根据不同的用户核对不同的使用权限,核对相符时才允许使用该文件。通常,文件主可对自己的文件执行一切操作。

Linux系统就采用这种方法,Linux系统规定用户使用文件的权限仅是读、写和执行3种,且相互间没有隐含关系。因此,用3位二进制数就能够表示一类用户对某个文件的存取权限,3类用户共需9位二进制数,从左往右每3位分别表示文件主、伙伴和一般用户的读、写和执行的权限。每一位的值为"1"时表示允许执行相应的操作,而为"0"时表示不允许执行该操作。文件主可以根据情况规定他的伙伴及一般用户对文件的使用权限。

6.8.3 文件的保密

文件的保密是防止不经文件拥有者授权而窃取文件。规定文件的使用权限在一定程度上可起到文件保密的作用,但是,文件的使用权限可由用户设定或修改,因而单靠规定文件的使用权限不能达到文件保密的目的。常用的文件保密的措施有以下几种。

1) 隐蔽文件目录

把保密文件的文件目录隐蔽起来,这些文件的文件目录不在显示器上显示。非授权的用户不知道这些文件的文件名,因而不能使用这些文件。IBM系列的操作系统中采用了这种方法,通过专用命令可以隐蔽和解除隐蔽指定的文件目录。

2) 设置口令

为文件设置口令是实现文件保密的一种可行方法,建立文件时把口令存放在文件目录中。用户使用文件时必须提供口令,仅当提供的口令与文件目录中的口令一致时,才可按规定的使用权限使用文件。为防止口令泄密,应采取隐蔽口令的措施,即在显示文件目录时应把口令隐藏起来。万一口令泄密,应及时更改口令。如果允许用户共享文件,可把口令通知授权用户,但当收回某个用户的使用权时必须更改口令,而更改后的口令又必须通知其他的授权用户。

3) 使用密码

对极少数极为重要的保密文件,可把文件信息转换成密码形式保存,使用文件时再将其解密。密码的编码方式只限文件主及允许使用文件的伙伴知道,这样,非授权用户就窃取不到文件信息。采用密码的方法增加了文件重新编码和解密的工作,使系统的开销增大。

6.9 文件的层次模型

本节介绍文件系统的一般层次模型,以便使读者对文件系统形成一个完整的概念。操作系统层次结构的设计方法是Dijkstra于1967年提出的,1968年Madnick将这一思想引

入了文件系统。层次结构的优点是,可以按照系统所提供的功能来划分为各种不同的层次,下层为上层提供服务,上层使用下层的功能。这样,上、下层之间彼此无须了解对方的内部结构和实现方法,而只关心二者的接口。从而,一个看上去十分复杂的系统将会由于层次的划分而变得易于设计、理解和实现。而且,当系统出现错误时,也容易进行查错和调整。因此,层次化设计方法也使得系统的管理和维护更加容易。

不过,在层次化设计方法中,层次的划分是一个十分复杂的问题。如果层次划分太少,则每一层的内容仍然十分复杂,分层的意义不明显;若层次划分太多,则各层之间传递的参数会急剧增加,而且每一层的处理会占去一定的系统开销,从而影响系统效率。因此,层次的划分要根据实际需要仔细地考虑。Madniek 把文件系统划分为 8 层,如图 6.23 所示。

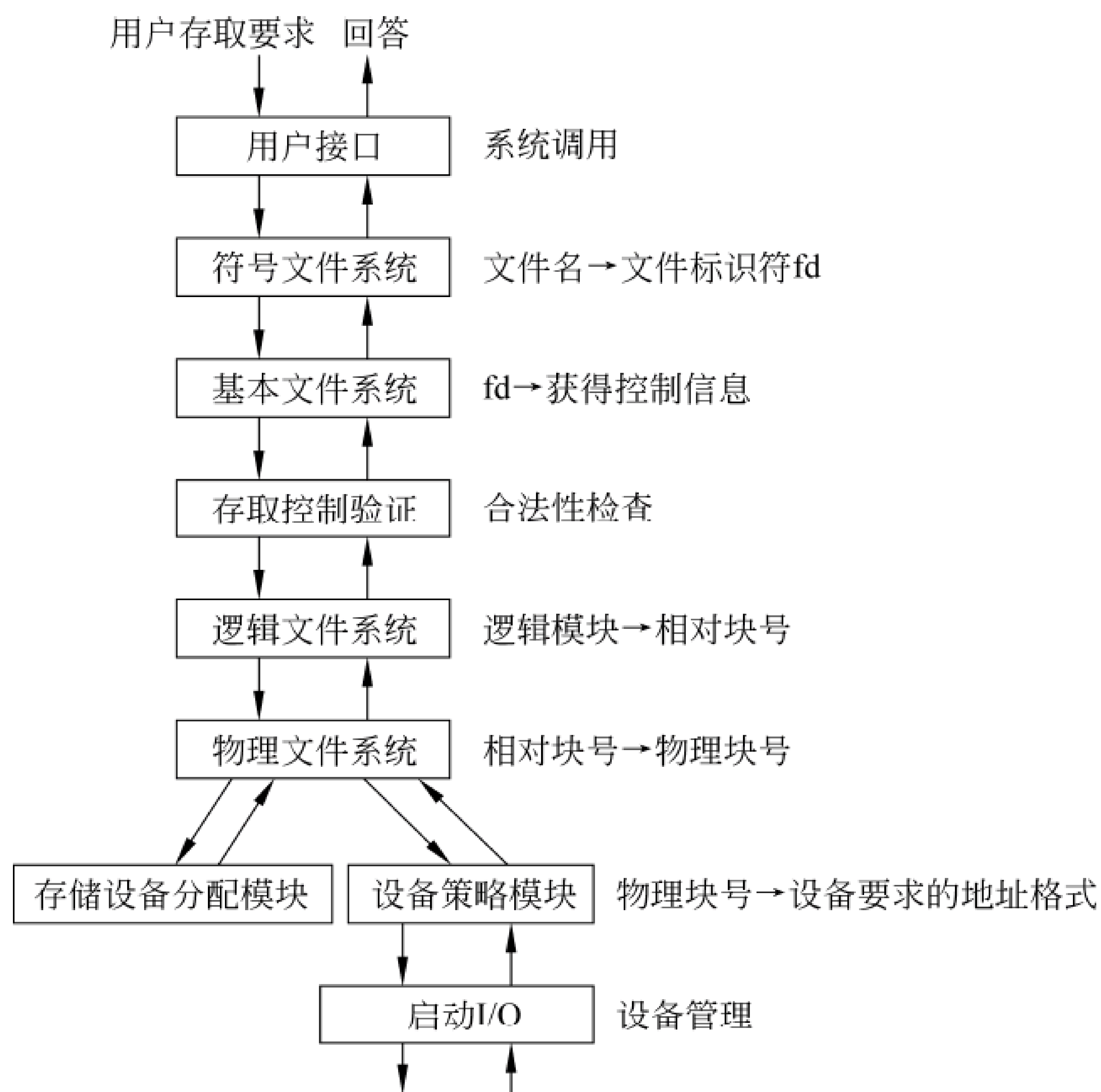


图 6.23　文件系统的层次模型

在图 6.23 中,第 1 层是用户接口,该层根据用户对文件的存取要求,把不同的系统调用加工改造成不同的内部调用格式。

第 2 层符号文件系统层。该层完成第 1 层所要求的功能,并把第 1 层所提供的参数——用户文件名转换成系统内部的唯一标识符 fd。该层的主要工作是搜索文件目录,也就是搜索 SFD,以找到相应文件名的表目并找到 fd。然后,fd 将作为参数传给第 3 层。

第 3 层是基本文件系统层。该层根据第 2 层的调用参数 fd,找到文件的说明信息,包括存取控制矩阵、文件逻辑结构、物理结构以及第一个物理块地址等。

第 4 层是存取控制验证层。该层的主要功能是根据存取控制信息和用户访问要求,检验文件访问的合法性,从而实现文件的共享、保护和保密。

第 5 层是逻辑文件系统层。该层的主要功能是根据文件的逻辑结构,找到所要进行操

作的数据或记录的相对块号。对于字符流的无结构文件来说,只要把用户指定的逻辑地址按块长换算成相对块号就可以了。但是,对于记录式有结构文件来说,由于用户有时指定的是键名或记录名,因此,需首先由键名(或记录名)搜索到相应的记录并得到对应的逻辑地址之后,再将其转换为相对块号。

第 6 层是物理文件系统层。该层根据文件的物理结构把相对块号转换成物理地址。

第 7 层是文件存储设备分配模块和设备策略模块。文件存储设备分配模块实现对空闲存储块的管理,包括分配、释放和组织。设备策略模块主要是把物理块号转换成相应文件存储设备所要求的地址格式,例如磁盘的柱面号、磁道号和扇区号等。然后,根据具体的操作要求及必要的参数,准备启动输入输出设备的命令。

第 8 层是启动输入输出层。由设备处理程序执行具体的读或写文件操作。

第 7 层和第 8 层是文件系统和设备管理程序的接口层。

6.10 Linux 的文件管理

Linux 为了支持多种不同的文件系统,引入了纯软件中间层——虚拟文件系统(Virtual File System,VFS),使文件子系统的可扩展性和可维护性变得更好。本节先介绍 VFS 的运作原理,然后再介绍一个物理文件系统——EXT2。

6.10.1 虚拟文件系统(VFS)

1. VFS 的作用

Linux 支持多种文件系统,如 EXT2、VFAT 和 IS09660 等。VFS 是内核软件层。它为用户空间的程序提供了诸如 open()、read()之类的统一编程接口,同时屏蔽了不同文件系统之间的差别,如图 6.24 所示。

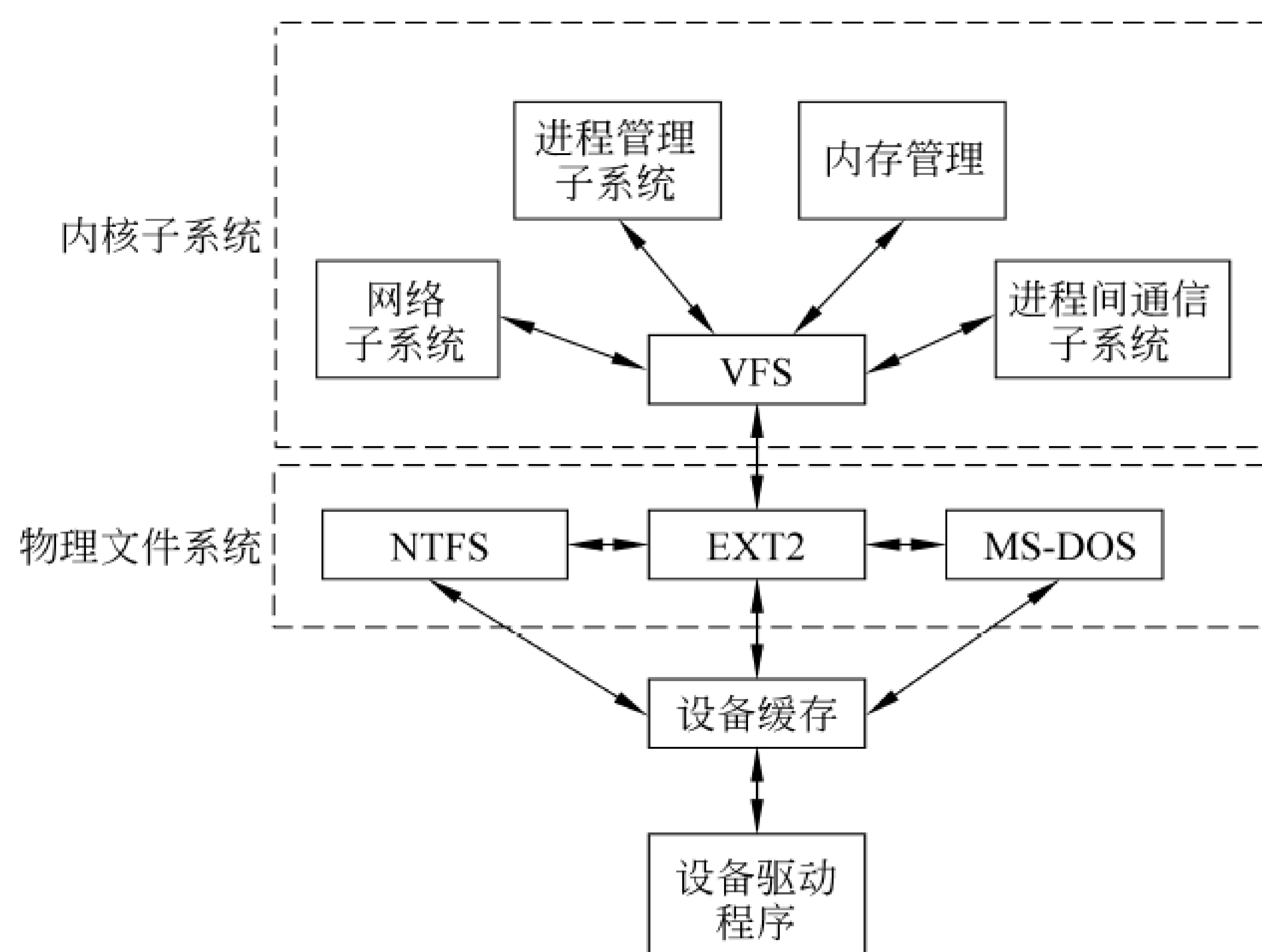


图 6.24 VFS 屏蔽了物理文件系统之间的差异

之所以称它为虚拟,是因为该文件系统的各种数据结构都是随时建立或删除的,在盘上并不永久存在,只能存在于内存;只有 VFS 是无法工作的,它不是一个真正的文件系统。

2. VFS 中的主要数据结构

1）超级块

超级块存储已安装文件系统的信息，通常对应磁盘文件系统的文件系统控制块。

```
struct super_block{
    struct list_head s_list;                /* 将所有的超级块链接起来 */
    kdev_t s_dev;                           /* 所在设备号 */
    unsigned long s_blocksize;              /* 该文件系统磁盘块的大小(字节数) */
    unsigned char s_blocksize_bits;         /* log2(s_blocksize) */
    struct file_system type * s_type;       /* 所属的文件系统类型 */
    struct super_operations * s_op;
    struct dentry * s_root;                 /* 文件系统的根目录 dentry 对象 */
    struct list_head s_dirty;               /* 修改过的 inode 队列 */
};
```

2）inode

inode 存储某个文件的信息。在文件系统内有唯一的 inode 号。通常对应磁盘文件系统的文件控制块。struct inode 结构的部分成员如下：

```
struct inode{
    struct list_head i_hash:                /* hash 值相同的 inode 链表 */
    struct list_head i_dentry;              /* 同属一个 inode 的 dentry 对象链表 */
    unsigned long i_ino;                    /* inode 号 */
    kdev_t i_dev;                           /* 所在设备的设备号 */
    umode_t i_mode;                         /* 表示文件类型及权限 */
    uid_t i_uid;                            /* 文件拥有者的用户 ID */
    gid_t i_gid;                            /* 用户所在组的 ID */
    loff_t i_size;                          /* 文件大小 */
    time_t i_atime;                         /* 最近一次的访问时间 */
    time_t i_mtime;                         /* 最近一次的修改时间 */
    time_t i_ctime;                         /* 文件创建时间 */
    struct inode_operations, * i_op;
    struct super_block * i_sb;              /* 所属的超级块 */
    struct addressspace * i_mapping;
};
```

3）文件

文件存储一个打开的文件和一个进程的关联信息。只要文件一经打开，这个文件就一直存在。文件用 struct file 结构描述：

```
struct file{
    struct list_head f_list;
    struct dentry * f_dentry;               /* 指向与文件对象关联的 dentry 对象 */
    struct vfsmount * f_vfsmnt;
    struct file_operations * f_op;          /* 文件对象的操作集合 */
    unsigned int f_flags;                   /* 使用 open()时设定的标志 */
```

```
    mode_t f_mode;
    loff_t f_pos;                          /* 对文件读写操作的当前位置 * /
    struct fown_struct f_owner;
};
```

files_struct 结构描述被进程打开使用的文件信息，fd 成员指向一个 struct file 指针数组，数组中的已使用项指向某个正被打开的文件对象，数组中的索引称为文件号，用户程序就是使用文件号对打开的文件进行操作。fd 开始指向 fd_array 数组，进程打开的文件超过 32 个时，会重新分配一个更大的 struct file 指针数组并让 fd 指向它，前提当然是进程被允许打开更多的文件。

```
struct files_struct{
    struct file * * fd;
    struct file * fd_array[NR_OPEN_DEFAULT];        /* NR_OPEN_DEFAULT 为 32 * /
};
```

4）dentry

dentry 主要是描述文件名及其相关联的 inode 信息。

```
struct dentry{
    struct inode * d_inode;                /* 该 dentry 所属的 inode * /
    struct dentry * d_parent;              /* 父目录 * /
    struct list_headd_hash;                /* hash 值相同的 dentry,链表 * /
    struct list_headd_subdirs;             /* 子目录链表 * /
    struct qstrd_name;                     /* 文件名及附属信息 * /
    struct dentry_operations * d_op;
};
```

inode 与文件之间的关系是一对一的。而一个真正的文件可能有多个文件名，比如硬链接（hard link），因此 dentry 与 inode 之间是多对一的关系。不同的进程可能打开同一个文件，但却不能用同一个文件来描述，因为操作标志、文件操作的位置等均可能不同，因而文件对象与 dentry 对象之间是多对一的关系。

5）super_operations

super_operations 结构的成员如下：

```
struct super_operations{
    void(* read_inode)(struct inode * );
    void(* write_inode)(struct inode * ,iht);
    void(* put_inode)(struct inode * );
    void(* delete_inode)(struct inode * );
    void(* put_super)(struct super_block * );
    void(* write_super)(struct super block * );
    …
};
```

read_inode：从已装载（mount）的文件系统读入一个 inode 的信息。该 inode 对象的 inode 号已事先被初始化。

write_inode：把 inode 信息写入磁盘。

put_inode：从 inode cache 移去 inode 对象。

delete_inode：删除 inode。

put_super：卸载(unmount)时 VFS 释放超级块。

write_super：把 VFS 超级块写入磁盘。

6）inode_operations

inode_operations 结构的成员如下：

```
struct inode_operations{
    int(* create)(struct inode* ,struct dentry* ,int);
    struct dentry* (* lookup)(struct inode* ,struct dentry* );
    int(* link)(struct dentry* ,struct inode* ,struct dentry* );
    int(* unlink)(struct inode* ,struct dentry* );
    int(* symlink)(struct inode* ,struct dentry* ,const char* );
    int(* mkdir)(struct inode* ,struct dentry* ,int);
    int(* rmdir)(struct inode* ,struct dentry* );
};
```

create：第一个参数必须是目录型文件对应的 inode。create 在该目录下创建一个文件，文件名事先被置入第二个参数中。第三个参数是文件的创建类型。

lookup：第一个参数必须是目录型文件对应的 inode。lookup()在该目录下查找一个文件，文件名事先被置入第二个参数 dentry 中。查找到的信息填入 dentry 中并返回。

7）file_operations

file_operations 结构的成员如下：

```
struct file_operations{
    loff_t(* llseek)(struct file* ,loff_t,int);
    ssize_t(* read)(struct file* ,char* ,size_t,loff_t* );
    ssize_t(* write)(struct file * ,const char* ,size_t,loff_t* );
    int(* mmap)(struct file* ,struct vm_area_struct* );
    int(* open)(struct inode* ,struct file* );
    ...
};
```

这些操作与系统调用接口非常相似，因此不再介绍。

3. 文件系统的注册与装载

1）文件系统的注册

每一种文件系统使用前必须先注册一个 file_system_type 对象。其成员如下：

```
struct file_system_type{
    const char * name;                          /* 文件类型名 * /
    struct super_block * (* read_super)(struct super_block* ,void * ,int);
    struct file_system_type * next;
};
```

2）文件系统的装载

一个文件系统只进行注册是不行的，还必须把它装载到某个目录下。用户经常使用 mount 命令把一个分区装载到某个目录下。内核与之对应的系统调用是 sysmount()。

虚拟文件系统执行装载命令的过程比较复杂，在此不进行说明。

3）文件系统的卸载

如果文件系统中的文件正在使用，该文件系统不能被卸载。如果文件系统中的文件或目录正在使用，VFS 索引节点高速缓存中可能包含相应的 VFS 索引节点，检索代码在索引节点高速缓存中根据文件系统所在设备的标识符查找是否有来自该文件系统的 VFS 索引节点，如果有且使用计数大于 0，则说明该文件正在使用，因此该文件系统不能被卸载；否则查看对应的 VFS 超级块，如果该文件系统的 VFS 超级块标志为“脏”，则必须将超级块信息写回磁盘。上述过程结束之后，对应的 VFS 超级块被释放。

用户常使用 unmount 命令把已装载的文件系统从系统中卸载。

6.10.2　EXT2 文件系统

EXT2 文件系统可能是 Linux 使用最广泛的文件系统。它支持传统 UNIX 文件的语义及一些高级特性，在性能和健壮性方面都表现不错。

1. EXT2 在磁盘上的物理布局

EXT2 分区的第一个磁盘块用于引导，其余的部分被分成组，如图 6.25 所示。所有的组大小相同并且顺序存放，所以由组的序号可以确定组在磁盘上的位置。

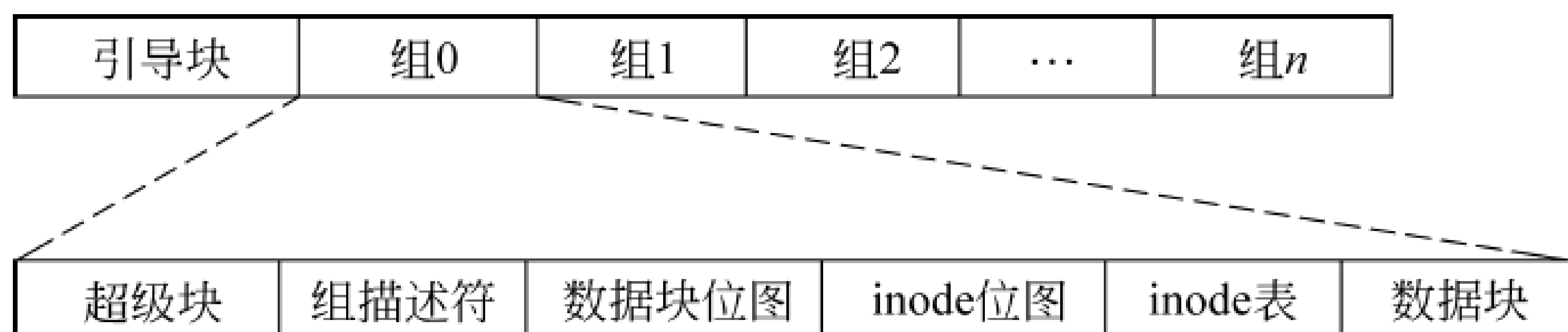


图 6.25　EXT2 分区的物理布局

每个组由如下部分组成：

（1）文件系统的超级块；

（2）所有组的描述符；

（3）块的位图；

（4）inode 位图；

（5）inode 表；

（6）数据块。

每个组都有一份文件系统的超级块和所有组的描述信息副本，正常情况下内核只有使用第 0 组的超级块和所有组的描述信息，并在合适时机让各个组的副本一致。当组 0 的副本遭到破坏时，便可根据其他组的副本恢复，从而加强文件系统的可靠性。

而块的位图、inode 位图、inode 表和数据块则专属于组。块位图的大小为一块，位图的每一位顺序对应组中的一块，0 表示块可用，1 表示已占用。如果块的大小为 1KB，则每组的大小为 8196KB。所有的文件及目录都需要用 inode 结构表示，inode 表用来存放这样的数据。inode 位图用来表示对应的 inode 表的空间是否已被占用。

2. 主要的数据结构及基本操作

1）超级块

超级块所用的数据结构是 ext2_super_block 结构，具体内容如下：

```
struct ext2_super_block{
    u32 s_inodes_count;              /* inode 总数 */
    u32 s_blocks_count;              /* 块总数 */
    u32 s_free_blocks_count;         /* 空闲块总数 */
    u32 s_free_inodes_count;         /* 空闲 inode 总数 */
    u32 s_first_data_block;          /* 第一个数据块 */
    u32 s_log_block_size;            /* 块的大小 */
    u32 s_blocks_per_group;          /* 每组块数 */
    u32 s_inode_per_group;           /* 每组 inode 数 */
    u32 s_mtime;                     /* 安装时间 */
    u32 s_wtime;                     /* 写时间 */
    u16 s_mnt_count;                 /* 安装次数 */
    …
}
```

2）描述符

每一组都有自己的描述符，描述符的结构如下：

```
struct ext2_group_desc{
    u32 bg_block_bitmap;             /* 本组块位图所在的块号 */
    u32 bg_inode_bitmap;             /* 本组 inode 位图所在的块号 */
    u32 bg_inode_table;              /* 本组 inode 表的起始块号 */
    u16 bg_free_block_count;         /* 组中空闲块的数目 */
    u16 bg_free_inode_count;         /* 组中空闲 inode 的数目 */
    u16 bg_used_dirs_count;          /* 组中目录的数目 */
}
```

3）inode

EXT2 文件系统所有的 inode 大小为 128B，inode 在 inode 表中依次存放，每个 inode 有一个预定的 inode 号。inode 的结构如下：

```
struct ext2_inode{
    u16 i_mode;                      /* 文件类型和访问权限 */
    u16 i_uid;                       /* 拥有者的 ID */
    u32 i_size;                      /* 文件大小 */
    u32 i_atime;                     /* 最后一次访问时间 */
    u32 i_ctime;                     /* 创建时间 */
    u32 i_mtime;                     /* 最后一次修改时间 */
    u16 i_gid;                       /* 组 ID */
    u32 i_blocks;                    /* 磁盘块数 */
    u32 i_block[EXT2_N_BLOCKS];      /* 指向磁盘块的指针 */
    …
}
```

EXT2_N_BLOCKS 的默认值为 15。

i_block[15]数组主要是支持常规文件，因为数据在磁盘上并不一定连续，需要保存各个磁盘块号。它的前 12 项可看成一级指针，直接存放文件数据所在的磁盘块号。数组的第 13 项是一个二级指针，指向的磁盘块并不包含文件的数据，而是一系列的一级指针，这些一级指针才用来指向磁盘块。数组的第 14、15 项分别是三级指针和四级指针，访问数据方式可由第 13 项类推。这种方法保证了对大量的小文件访问效率高，同时又支持大文件，如图 6.9 所示。

因为每组磁盘块的 inode 数目固定，所以在知道文件的 inode 号的情况下很容易计算出该文件属于哪个组并且得到 inode 在组表中 inode 的下标，继而可得到 ext2_inode 信息。

3. 磁盘块的分配与释放

磁盘块的释放主要工作是修改块位图，并且涉及块的统计变量。

磁盘块的分配稍为复杂。分配算法先试图分配那些与上次分配给文件的块连续的空间；如果不行，则试图在附近 32 个块的范围内分配；若还不行，则在本组内向前找 8 个连续空闲的块；若还不满足，则任何空闲的块均可以被分配；若还不满足，则到其他的组中去寻找。这样做的目的是使文件所占的存储空间尽可能地连续分配，使文件访问时间尽量变短。

6.11 本章小结

本章首先介绍了文件的概念、文件的分类和文件系统的概念。

文件的组织方式有两种：逻辑结构和物理结构。

常用的逻辑结构有两种：流式结构（以字符为操作单位）和记录式结构（以记录为操作单位）。常有的记录式结构有 4 种：连续结构、多重结构、转置结构和顺序结构。常用的存取方法有顺序存取法、随机存取法（直接存取法）和按键存取法。

存储在顺序存储介质（磁带）上的文件通常是顺序文件，每个文件都有文件头标、文件信息和文件尾标 3 个组成部分。

存储在随机存储介质（磁盘）上的文件的结构有顺序结构、链接结构和索引结构。

把文件的逻辑结构转成物理结构和把文件的物理结构转成逻辑结构是文件系统的任务之一。由于逻辑单位与物理单位不同，在把逻辑文件以物理文件方式存储在存储介质上时，为了提高效率，对记录进行成组；在把物理文件从存储介质上读出转成逻辑文件时就要对记录进行分解。

文件目录是实现按名存取和对文件进行管理的一种数据结构。文件目录的组织方式有一级目录、二级目录和树形目录。系统中的文件目录以文件的方式存储在磁盘上（目录文件）；为了节省内存空间，也为了提高操作速度，只把目录文件中与当前目录有关的部分读入内存。

对磁盘空间的管理方法有位示图、空闲块表和空闲块链。

为了提高存储系统的可靠性，常采用的技术有：第一级容错技术、第二级容错技术、廉价磁盘冗余阵列和后备系统。

对文件进行的主要操作有建立文件、打开文件、读文件、写文件、关闭文件和删除文件 6 种操作。

实现文件共享的方式有绕道法、链接法和基本文件目录表。

对文件进行保护主要是为了防止两方面的破坏：防止系统故障造成的破坏（主要措施有建立副本和定时转储）和防止用户共享文件可能造成的破坏（主要措施有采用树形目录结构、存取控制矩阵和文件使用权限）。

对文件保密的方法有隐蔽文件目录、设置口令和使用密码。

Madniek 把文件系统划分为 8 层，下层为上层提供服务，上层使用下层的功能，这样使十分复杂的系统由于层次的划分而变得易于设计、理解和实现。

最后简单地介绍了 Linux 的文件系统，主要有 VFS 和 EXT2。

习　　题

1. 什么是文件和文件系统？文件是怎样分类的？
2. 简述文件系统的特点和功能。
3. 文件有几种组成结构？各有哪些特点？
4. 逻辑文件有哪几种结构？各种结构有何特点？
5. 记录文件有哪几种结构？各种结构有何特点？常用的存取方法有哪几种？
6. 说明对记录的搜索的主要内容。常用的搜索方法有哪几种？
7. 磁带上的文件由哪些部分组成？简述查找磁带上的文件的过程。
8. 磁盘上的文件有哪几种组织方式？各有哪些特点？
9. 什么是记录的成组和分解？记录的成组有何意义？何谓块因子？
10. 什么是文件目录？它包含哪些信息？
11. 有哪几种文件目录结构？各有何特点？文件目录是如何管理的？
12. 磁盘存储空间是如何管理的？各种管理方法有何特点？
13. 磁盘容错技术分为哪几级？各级包含哪些措施？
14. 什么是 RAID？它有何优点？
15. 对文件主要进行哪些操作？
16. 如何实现文件的共享？
17. 文件保护的含义是什么？造成文件被破坏的主要因素有哪些？采取怎样的措施防止文件被破坏？
18. 文件保密的含义是什么？文件保密的措施主要有哪些？
19. 在 Linux 中为何引入 VFS？它的通用文件模型由哪几部分组成？
20. 在 EXT2 分区中，组由哪些部分组成？主要的数据结构有哪些？
21. 设用户 A 有名为 W1、W2 和 W3 的 3 个私有文件，用户 B 有名为 J1 和 J2 的两个私有文件，这两个用户都需使用共享文件 T。文件系统对所有用户提供按名存取功能，为保证存取的正确性，文件系统应设置合理的目录结构，请画出该目录的结构。
22. 假定磁带的记录密度为每英寸 800 个字符，每一记录长度为 160 个字符，块与块之间的间隙为 0.6 英寸，现有 1000 条逻辑记录需要存放在磁带上，分别回答下列问题：

(1) 计算不采用成组操作时的磁带空间利用率。

（2）计算采用以5条记录为一组的成组操作时的磁带空间利用率。

（3）为了使磁带空间利用率大于50%，采用成组记录时块因子最少为多少？

23. 假定文件系统把文件存储在磁盘上时采用链接结构，磁盘的分块大小为512B，而逻辑记录的大小为250B。现有一个名为ABC的文件，共有10条逻辑记录，回答下列问题：

（1）怎样才能有效地利用磁盘空间？

（2）画出文件ABC在磁盘上的链接结构(磁盘块号自定)。

（3）若用户要求读包含第1453个字符的逻辑记录，请写出读过程的主要工作步骤。

24. 有一个可以带4台终端的计算机系统，该系统配置了一个磁盘存储终端用户的程序和数据。今有4名上机实习的学生，他们在各自的终端上输入自己的程序和数据，并都保存在磁盘上，凑巧他们给各自的程序取的文件名均为JOU，请问：

（1）系统应采用怎样的目录结构才能区别这些学生的程序？画出这个目录结构。

（2）简单阐明系统怎样为这4名学生索取他们各自的程序。

25. 某文件系统的结构如图6.26所示，其中方框代表目录，圆圈代表文件，请回答下列问题：

（1）如果用“/”分隔各个目录和文件，用“..”表示一个目录的父目录，设当前目录为P，写出访问文件j的路径。

（2）将目录N改为P是否可以？将目录T改为U是否可以？为什么？

（3）将文件b改为c是否可以？将文件d改为e是否可以？为什么？

（4）将文件d改为N是否可以？为什么？

（5）当前目录是O，写出对文件a的共享方法。

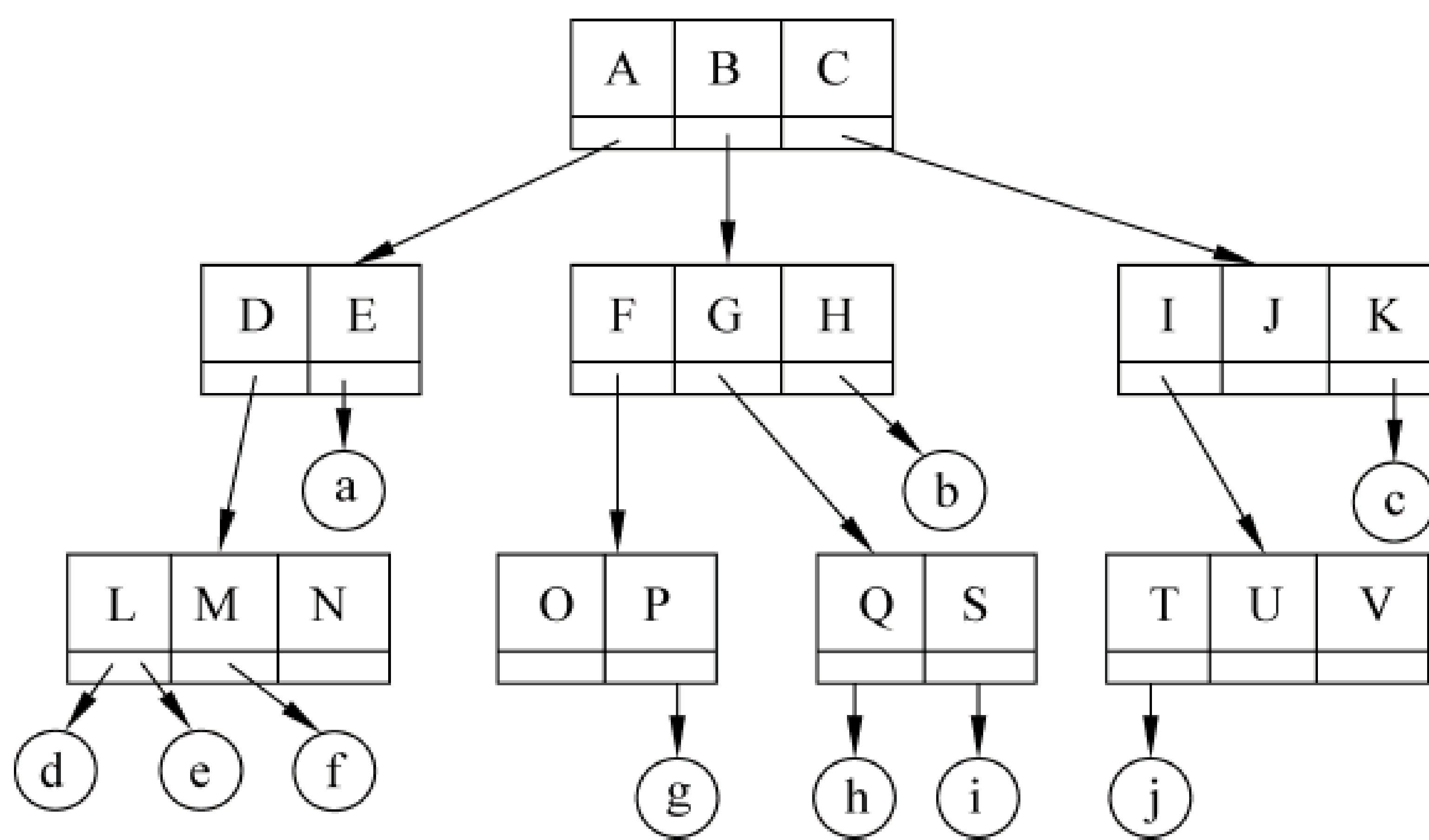


图6.26　某文件系统的结构

26. 若文件被误删，说明在什么情况下可以恢复，恢复的原理是什么？

27. 某文件系统为一级目录结构，文件的数据一次性写入磁盘，写入的文件不可修改，但可多次创建新文件。请回答如下问题：

（1）在连续、链式和索引3种文件的数据块组织方式中，哪种更合适？要求说明理由。为定位文件数据块，需要在FCB中设计哪些相关描述字段？

（2）为快速找到文件，对于FCB而言，是集中存储好，还是与对应的文件数据块连续存储好？要求说明理由。

28. 文件系统中采用混合索引分配时，设块长为512B，每个块号长度为3B，如果不考虑逻辑块号占用的物理位置，分别求出采用两级索引和三级索引时可寻址的最大文件长度。

29. 设文件P由连续结构的定长记录组成，每个记录的长度为500B，每个物理块长1000B，且物理结构也为连续结构和采用直接存取方式；请按照图6.22所示的文件系统模型，写出系统调用read(P,5,15000)的各层执行结果。其中P为文件名，5为记录号，15000为内存地址。

第 7 章　设 备 管 理

设备管理是操作系统的重要组成部分之一,它负责管理和控制 I/O 设备。由于 I/O 设备种类繁多,与计算机连接方式复杂而多样,因而设备管理是操作系统中一个相当庞大复杂的功能模块。本章仅介绍设备管理的基本原理和方法。掌握了这些基本原理和方法,读者在了解具体的设备管理系统时就会容易得多。本章主要讨论设备管理的基本概念,包括中断、通道、缓冲、设备分配和 I/O 控制,以及最常用的存储设备——磁盘的驱动调度等。

7.1　设备管理概述

7.1.1　设备的类别

早期的计算机系统速度慢、应用面窄,外部设备主要以纸带、卡片等作为输入输出介质,相应的设备管理程序也比较简单。进入 20 世纪 80 年代以来,由于个人计算机、工作站以及计算机网络系统等的发展,外部设备开始走向多样化、复杂化和智能化。例如,在有的网络卡中就装有自己的 CPU(网络处理器),以处理网络上信息的输入输出。再者,以某种硬件设备为基础的虚拟设备和仿真设备技术也得到了广泛应用,例如虚拟终端技术和仿真终端技术等。近年来最为流行的窗口系统中的 X-Window 等都是作为一种设备和操作系统相连的。这使得设备管理变得越来越复杂。

如此众多的外部设备可以从不同角度进行分类。

按设备的使用特性,可将外部设备分为存储设备、输入输出设备、终端设备以及脱机设备等,如图 7.1 所示。

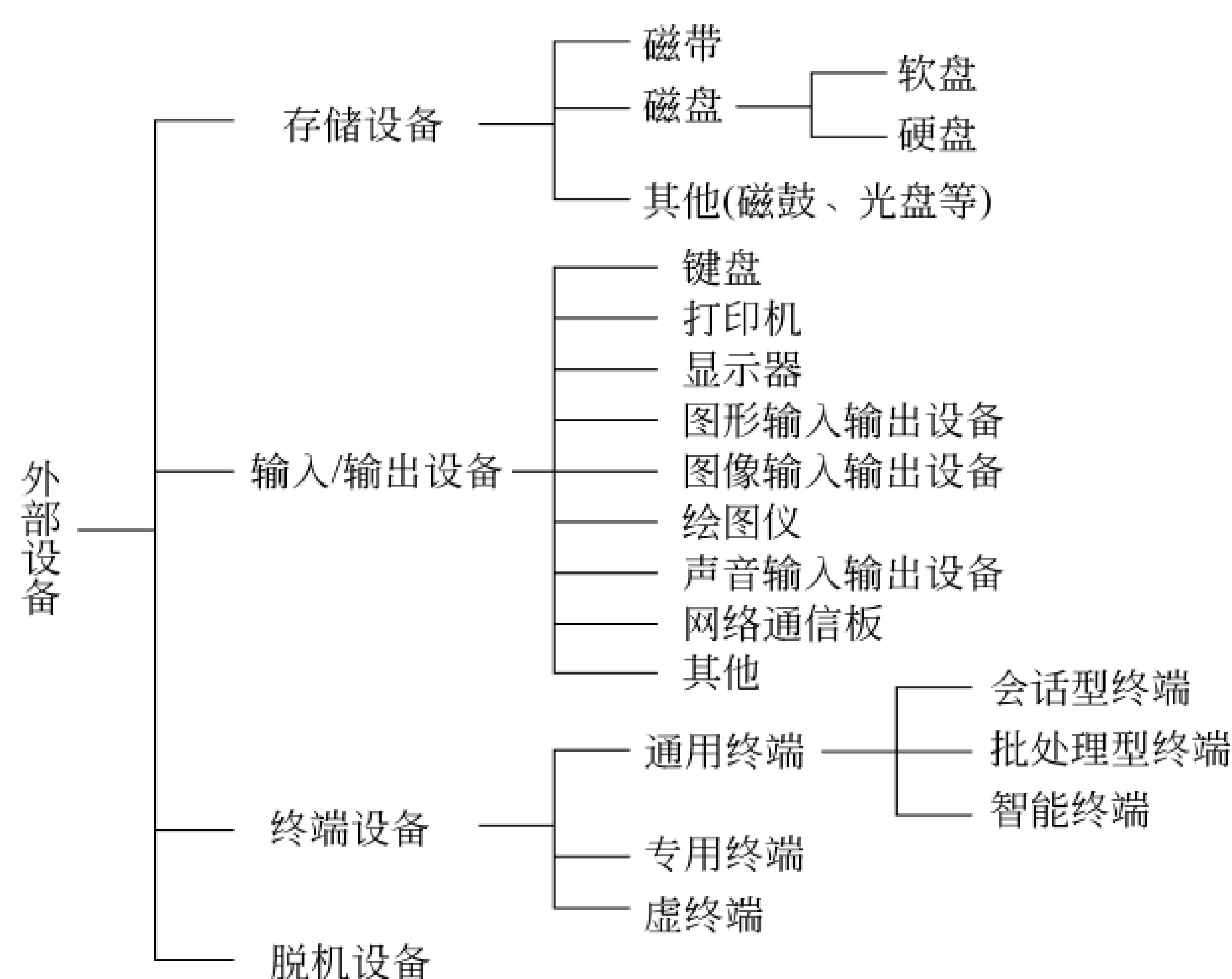


图 7.1　按使用特性对外部设备的分类

按设备的从属关系，可将外部设备划分为系统设备和用户设备。系统设备是指那些在操作系统生成时就已配置好的各种标准设备，例如键盘、显示器以及文件存储设备等。用户设备则是那些在系统生成时没有配置，而由用户自己安装配置后由操作系统统一管理的设备，例如网络系统中的各种网卡、实时系统中的A/D、D/A变换器、图像处理系统的图像设备等。

按信息交换的单位，可将外部设备划分字符设备和块设备。键盘、终端和打印机等以字符为单位组织和处理信息的设备被称为字符设备。字符设备的基本特征是其传输速率较低，通常为几个字节至数千字节；另一特征是不可寻址，即输入输出时不能指定数据的输入源地址及输出的目标地址；此外，字符设备在输入输出时，常采用中断驱动方式。而磁盘和磁带等以字符块为单位组织和处理信息的设备被称为块设备。磁盘的每个盘块的大小为512B～4KB，磁盘设备的基本特征是其传输速率较高，通常每秒钟为几兆位，另一特征是可寻址，即对它可随机地读/写任一块；此外，磁盘设备的I/O常采用DMA方式。

按传输速度的高低，可将外部设备分为低速设备、中速设备和高速设备。低速设备是指其传输速率仅为每秒钟几个字节至数百个字节的一类设备。属于低速设备的典型设备有键盘、鼠标器、语音的输入和输出等设备。中速设备是指其传输速率在每秒钟数千个字节至数十万个字节的一类设备。典型的中速设备有行式打印机和激光打印机等。高速设备是指其传输速率在数百个千字节至千兆字节的一类设备。典型的高速设备有磁带机、磁盘机和光盘机等。

按设备的共享属性，可将设备分为独占设备、共享设备和虚拟设备。独占设备是指在一段时间内只允许一个用户(进程)访问的设备，即临界资源。因而，对多个并发进程而言，应互斥地访问这类设备。系统一旦把这类设备分配给了某进程后，便由该进程独占，直至用完释放。应当注意，独占设备的分配有可能引起进程死锁。共享设备是指在一段时间内允许多个进程同时访问的设备。当然，对于每一时刻而言，该类设备仍然只允许一个进程访问。显然，共享设备必须是可寻址的和可随机访问的设备。典型的共享设备是磁盘。共享设备不仅可获得良好的设备利用率，而且它也是实现文件系统和数据库系统的物质基础。虚拟设备是指通过虚拟技术将一台独占设备变换为若干台逻辑设备用户(进程)同时使用。

对设备进行分类的目的在于简化设备管理程序。由于设备管理程序管理的是设备，是与硬件打交道的，不同的硬件特性不相同，因此，不同的设备硬件对应于不同的设备管理程序。对于同类设备，由于设备的硬件特性十分相似，从而可以利用相同的管理程序或只做很少的修改即可。

7.1.2 设备管理的功能和任务

设备管理是对计算机输入输出系统的管理，是操作系统中最具有多样性和复杂性的部分。

设备管理的主要任务如下：

(1) 选择和分配输入输出设备以便进行数据传输操作。

(2) 控制输入输出设备和CPU(或内存)之间交换数据。

(3) 为用户提供一个友好的透明接口，把用户和设备硬件特性分开，使得用户在编制应用程序时不必涉及具体设备，操作系统按用户要求控制设备工作。另外，这个接口还为新增

加的用户设备提供一个和操作系统核心相连接的入口,以便用户开发新的设备管理程序。

(4) 提高设备和设备之间、CPU 和设备之间以及进程和进程之间的并行操作程度,以使操作系统的效率提高。

为了完成上述主要任务,设备管理程序一般要提供下述功能:

(1) 提供和进程管理系统的接口。当进程要求设备资源时,该接口将进程要求转达给设备管理程序。

(2) 进行设备分配。按照设备类型和相应的分配算法把设备和其他有关的硬件分配给请求该设备的进程,并把未分配到所请求的设备或其他有关硬件的进程放入等待队列。

(3) 实现设备和设备、设备和 CPU 等之间的并行操作,这需要有相应的硬件支持。除了装有控制状态寄存器、数据缓冲寄存器等的控制器之外,对应于不同的输入输出(I/O)控制方式,还需要有 DMA(Directed Memory Access)、通道等硬件。

(4) 进行缓冲区管理。一般来说,CPU 的执行速度和访问内存速度都比较高,而外部设备的数据流通速度则低得多(例如键盘),为了减少外部设备和内存与 CPU 之间的数据传输速度不匹配的问题,系统中一般设有缓冲区(器)来暂时存放数据。设备管理程序负责进行缓冲区分配、释放及有关的管理工作。

7.1.3 数据传送控制方式

设备管理的主要任务之一是控制设备和内存或 CPU 之间的数据传送,这里介绍几种常用的数据传送控制方式。

外围设备和内存之间的常用数据传送方式有 4 种,即程序直接控制方式、中断控制方式、DMA 控制方式和通道控制方式。

1. 程序直接控制方式

处理器根据用户进程程序中的 I/O 语句(或指令),向 I/O 设备(或设备控制器)发出一个 I/O 命令,称为 I/O 操作。早期,外设只有设置忙/闲标志触发器 busy 的能力。如果设备闲置,则 busy=0;如果设备正在忙于输入输出,则 busy=1。与此相适应,CPU 配有一条 I/O 测试指令 test,用以测试 busy 的状态。以输入为例,设备自身有输入缓冲寄存器 in,读入的信息总是存放在 in 中。假定应把信息读到主存指定区域 inbuf,则 CPU 控制 I/O 的过程如下。

输入过程:

```
input: init(indev);                              //启动输入设备
       busy=1;
       IF test(busy) THEN Goto input;
       inbuf=(in);
       …
GOTOInput;
```

输出过程:

```
output: IF test(busy) THEN output;
        (out)=outbuf;
        init(outdev);                            //启动输出设备
```

```
        busy=1;
    GOTO output;
```

从上述过程可以看出,输入输出包含这样几步工作:启动、数据信息传输、I/O的管理(如计数、主存区域的指针控制等)及任务结束后的善后处理。从循环测试I/O方式可以看出,只要启动了输入输出,CPU大部分时间都耗费在等待数据信息传输完成的循环测试里,这无疑是一种极大的浪费。

2. 中断控制方式

在中断控制方式中,I/O操作由程序发起,在操作完成时(如数据可读或已经写入)由外设向CPU发出中断,通知该程序。数据的每次读写通过CPU。这种控制方式的优点是,在外设进行数据处理时,CPU不必等待,可以继续执行该程序或其他程序。其缺点是,CPU每次处理的数据量少(通常不超过几个字节),只适于数据传输率较低的设备。

在程序直接控制方式中,CPU不断测试busy标志触发器的目的,是要判定这一次输入输出是否完成。设备中断被引入后,外部设备具有了向CPU发送消息的能力,就能够通过中断来告知CPU这次I/O已经完成。因此用程序中断方式来控制设备的工作,CPU无须再去不断地测试busy的状态,可以从等待数据信息传输完成的循环动作中解脱出来。此时的输入输出,CPU就只负责启动、I/O的管理及整个任务结束后的善后处理。在数据信息传输时,表现为CPU与外设并行工作。"中断"使CPU与外设、外设与外设之间的并行工作成为可能。

3. DMA控制方式

由程序设置DMA控制器中的若干寄存器值(如内存始址和传送字节数),然后发起I/O操作,DMA控制器完成内存与外设的成批数据交换,在操作完成时由DMA控制器向CPU发出中断。这种控制方式的优点是:CPU只需干预I/O操作的开始和结束,而其中的一批数据读写无须CPU控制,适于高速设备。

4. 通道控制方式

通道又称为I/O处理机,它能完成主存和外设之间的数据传输,并与CPU并行操作。采用通道技术解决了I/O操作的独立性和各部件工作的并行性。通道把中央处理机从烦琐的I/O操作中解放出来。采用通道技术后,不仅能实现CPU和通道的并行操作,而且通道与通道之间也能实现并行操作,各通道上的外部设备也能实现并行操作,从而可达到提高整个系统的效率的根本目的。

对于上述4种控制方式进行选择和衡量应遵循的原则如下:

(1) 数据传送速度足够高,能满足用户的需要但又不丢失数据。

(2) 系统开销小,所需的处理控制程序少。

(3) 能充分发挥硬件资源的能力,使得I/O设备尽量忙,而CPU等待时间少。

7.2 磁盘的驱动调度

磁盘的驱动方式不同,从磁盘上获得信息或向磁盘上写信息的时间也不同,影响信息获取和存储的时间还有数据在磁盘上的分布方式。

7.2.1 磁盘的结构

磁盘机是一种高速、大容量、旋转型的存储设备,它能把信息记录在盘片上,也能把盘片上的信息读出。每个盘片有正反两面,若干张盘片可以组成一个磁盘组。一个磁盘组中的盘片都被固定在一个轴上,沿着一个方向高速旋转。每个盘面有一个读/写磁头,所有的读/写磁头被固定在唯一的移动臂上同时移动,把所有的读/写磁头按从上到下的次序从 0 开始进行编号,称为磁头号。每个盘面上有许多磁道,从 0 开始按由外向里的次序顺序编号,不同盘面上具有相同编号的磁道在同一个柱面上,这样把盘面上的磁道号称为柱面号。移动臂可以带动读/写磁头访问所有的磁道,当移动臂移动到某一位置时,所有的读/写头都在同一柱面上,每次只有其中的一个磁头可以进行读/写的操作。在磁盘初始化时把每个盘面划分成相等数量的扇区,按磁盘旋转的反向从 0 开始给各扇区编号,称为"扇区号"。每个扇区的各个磁道上均可存放相等数量的字符,称它为"块",块是信息读/写的最小单位。要确定一个块所在的位置,必须给出 3 个参数:柱面号、磁头号和扇区号。磁盘的结构如图 7.2 所示。

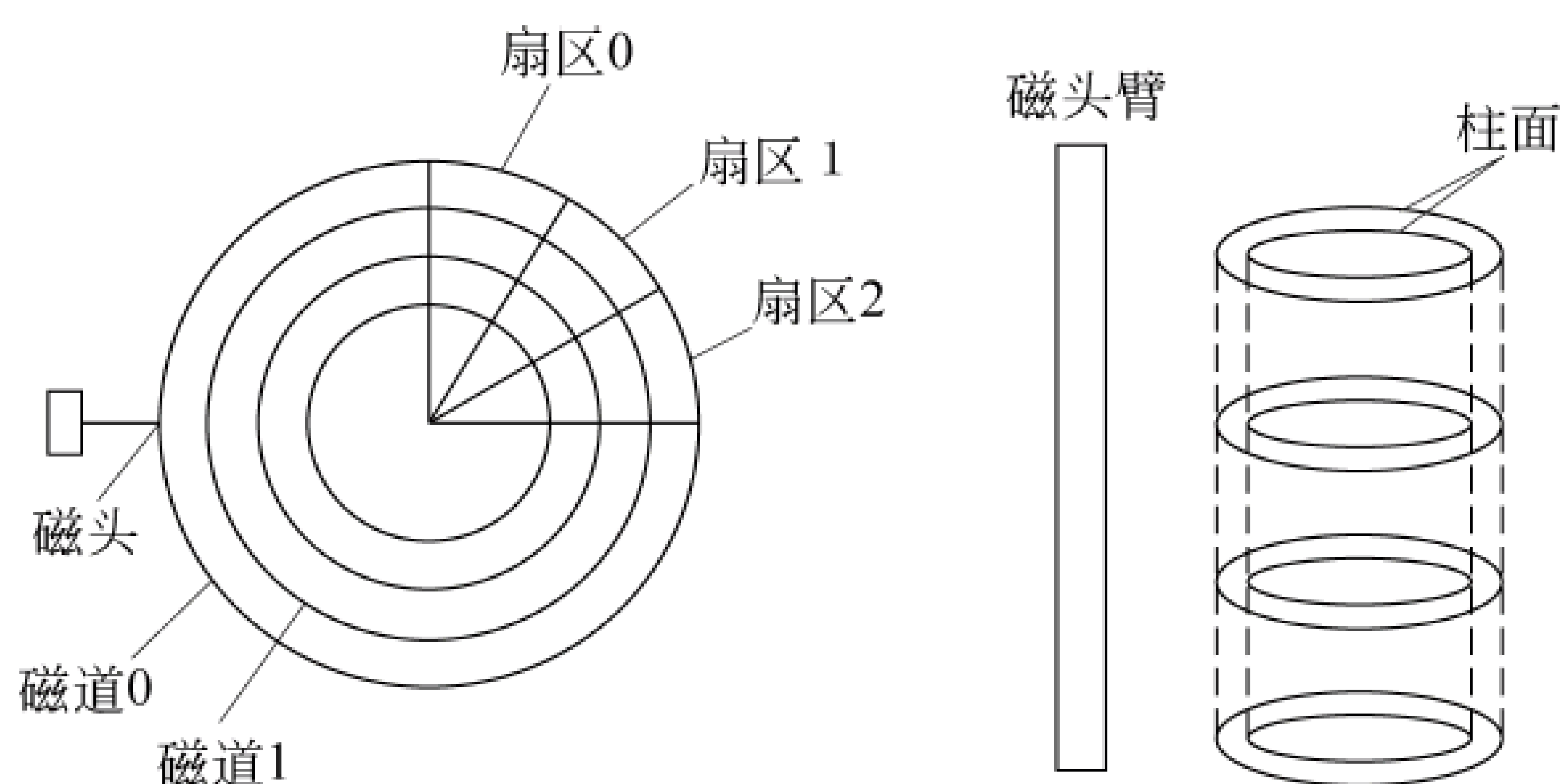


图 7.2 磁盘结构

启动磁盘执行 I/O 操作时,要把移动臂移动到指定的柱面上,再等待指定的扇区旋转到磁头位置下,然后让指定的磁头进行读/写,完成信息传送。因此,执行一次 I/O 所花的时间有:

(1) 寻找时间。磁头在移动臂带动下移动到指定柱面所花费的时间。

(2) 延迟时间。指定扇区旋转到磁头下所需的时间。

(3) 传送时间。由磁头进行读/写完成信息传送的时间。

其中传送信息所花的时间是硬件设计时就固定的,而寻找时间和延迟时间与信息在磁盘上的位置有关。图 7.3 是访问磁盘的操作时间示意。

为了减少移动臂进行移动花费的时间,每个文件的信息不是按盘面上的磁道顺序存放满一个盘面后,再放到下一个盘面上,而是按柱面存放,同一柱面上的各磁道被放满后,再放到下一个柱面上。所以,各磁盘块的编号按柱面顺序(从 0 号柱面开始),每个柱面按磁道顺序,每个磁道又按扇区顺序进行排序。假定用 T 表示每个柱面上的磁道数,用 S 表示每个盘面上的扇区数,则第 i 柱面、j 磁头、k 扇区所对应的磁盘块号 b 可由如下公式确定:

$$b = k + S \times (j + i \times T)$$

同样地,根据磁盘块号也可以确定该盘块在磁盘上的位置。在上述的假定下,每个柱面上有

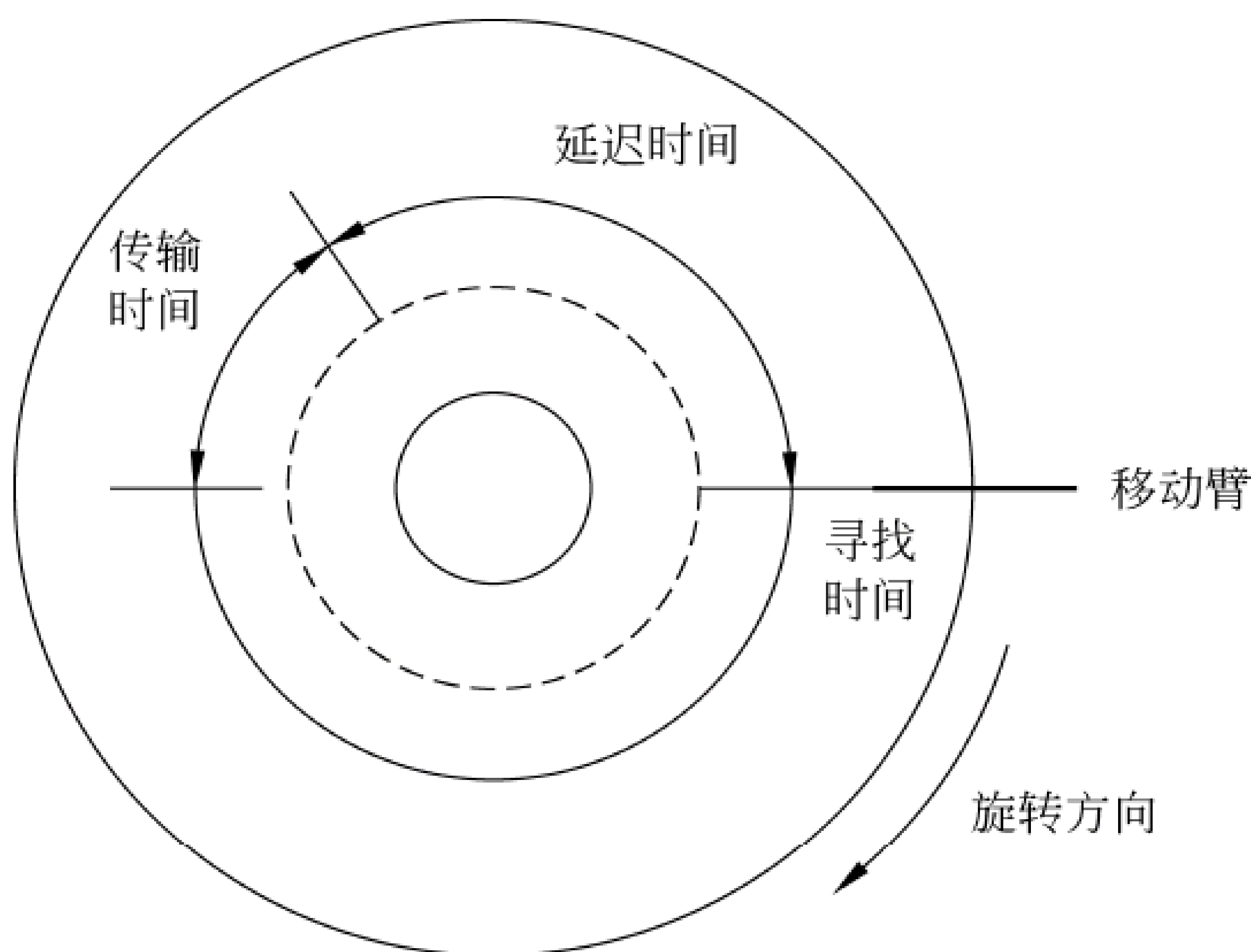


图 7.3 访问磁盘的操作时间

$S \times T$ 个磁盘块，为了计算第 P 块在磁盘上的位置，可以令 $D = S \times T$，设 $M = [P/D]$，$N = P \bmod D$。于是，第 P 块在磁盘上的位置为

$$柱面号 = M$$
$$磁头号 = [N/S]$$
$$扇区号 = N \bmod S$$

在微型计算机中，对软盘片来说。每个柱面只包含两个磁道。文件信息也是按柱面、磁道和扇区的顺序依次存放。

7.2.2 磁盘的驱动调度

磁盘是一种可共享的设备，在多道程序设计的系统中，同时会有若干个访问者请求磁盘执行 I/O 操作。但是，为了保证信息的安全，系统在每一时刻只允许一个访问者启动磁盘执行 I/O 操作，其余的访问者必须等待，一次 I/O 操作结束后才可让等待中的一个访问者启动磁盘。选择哪一个等待者去访问磁盘应根据移动臂的当前位置，使寻找时间和延迟时间尽可能小的那个访问者优先得到服务。这样，可以降低若干个访问者执行 I/O 操作的总时间，增加了单位时间内的 I/O 操作次数，有利于系统效率的提高。系统往往采用一定的调度策略来决定各等待访问者的执行次序，这项工作称为磁盘的驱动调度，采用的调度算法称为驱动调度算法。对磁盘来说，驱动调度由移臂调度和旋转调度两部分组成。

1. 移臂调度

根据访问者指定的柱面位置来决定执行次序的调度称为移臂调度，移臂调度的目的是尽可能地减少 I/O 操作中的寻找时间。常用的移臂调度算法有先来先服务算法、最短寻找时间优先算法、扫描调度算法、循环扫描调度算法、NStepSCAN 调度算法和 FSCAN 调度算法。

1）先来先服务调度算法(First Come First Served，FCFS)

先来先服务调度算法是最简单的移臂调度算法，它不考虑访问者要求访问的物理位置，而只是考虑访问者提出访问请求的先后次序。例如，假设柱面的编号为 0～199，如果现在读/写磁头正在 53 号柱面上执行 I/O 操作，而等待访问者依次要访问的柱面为 98，183，37，122，14，124，65，67，那么，当 53 号柱面上的操作结束后，访问柱面的次序为 98，183，37，

122,14,124,65,67,读/写磁头总共移动了 640 个柱面。

采用先来先服务调度算法决定等待访问者执行 I/O 操作的次序时,移动臂将来回地移动,读/写磁头总共移动的距离较长。先来先服务调度算法花费的寻找时间较长,因而执行 I/O 操作的总时间也很长。

2) 最短寻找时间优先调度算法(Shortest Seek Time First,SSTF)

最短寻找时间优先调度算法总是从等待访问者中挑选寻找时间最短的那个请求先执行,而不管访问者到达的先后次序。下面还是用同一个例子来讨论,现在当 53 号柱面的操作结束后,访问次序为 65,67,37,14,98,122,124,183,读/写磁头总共移动了 236 个柱面。

采用最短寻找时间优先调度算法决定等待访问者执行 I/O 操作的次序时,读/写磁头总共移动的距离较短。与先来先服务调度算法比较,大幅度地减少了寻找时间,因而缩短了为各访问者服务的平均时间,也就提高了系统效率。

3) 扫描调度算法(SCAN)

SSTF 调度算法能获得较好的寻道性能;但在该调度算法调度下,只要不断有新进程的请求到达,且该请求所要访问的柱面与磁头当前所在的柱面的距离较近,该新进程的请求必然优先满足,这样可能导致某个进程发生“饥饿”。对 SSTF 调度算法略加修改后形成扫描调度算法,即可防止老进程出现“饥饿”现象。

扫描调度算法总是从移动臂当前位置开始沿着移动臂的移动方向去选择离当前移动臂最近的待访问柱面进行访问。如果沿移动臂的移动方向无请求访问的柱面时,就改变移动臂的移动方向再选择最近的待访问柱面进行访问。移动臂移动如同电梯运作,故该调度算法也称电梯调度算法。

仍用上面的例子来讨论采用扫描调度算法的情况。由于该算法与移动臂的方向有关,所以应分两种情况来进行。

(1) 移动臂是向外移的。移动臂由里向外(向柱面号减小方向移动)到达 53 号柱面的位置,因此,当访问 53 号柱面的操作结束后,依次访问的次序为 37,14,65,67,98,122,124,183,读/写磁头共移动了 208 个柱面的距离。

(2) 移动臂是向里移的。移动臂由外向里(向柱面号增大方向移动)到达 53 号柱面的位置,因此,当访问 53 号柱面的操作结束后,依次访问的次序为 65,67,98,122,124,183,37,14,读/写磁头共移动了 299 个柱面的距离。

扫描调度算法与最短寻找时间优先调度算法都是尽量减少移动臂移动所花的时间,所不同的是最短寻找时间优先调度算法不考虑臂的移动方向,总是选择离当前读/写磁头最近的那个柱面的访问者,这种选择可能导致移动臂经常改变移动方向。扫描调度算法是沿着臂的移动方向去选择离当前读/写头最近的那个柱面的访问者,仅当沿臂移动方向无等待访问者时才改变臂的移动方向。由于移动臂改变方向是机械动作,速度相对较慢,相比之下,扫描调度算法是一种简单、实用且高效的调度算法。但是,在实现时除了要记住读/写磁头的当前位置外,还必须记住移动臂的移动方向。

4) 循环扫描调度算法(Circle SCAN,CSCAN)

SCAN 算法既能获得较好的寻道性能,又能防止“饥饿”现象,故被广泛用于大、中、小型计算机和网络中的磁盘调度。但 SCAN 也存在以下问题:当磁头刚从里向外移动而越过了某一磁道时,恰好又有一个进程请求访问此磁道,这时,该进程必须等待,待磁头继续从里向

外，然后再从外向里扫描完所有要访问的磁道后，才处理该进程的请求，致使该进程的请求被大大地推迟。为了减少这种延迟，CSCAN 算法规定磁头单向移动，不考虑等待访问者的先后次序，总是从最外面的待访问的柱面开始向里扫描，按照各访问者所要访问的柱面位置的次序去选择访问者。移动臂到达最后一个柱面后，立即带动读/写磁头快速返回到最外面的待访问的柱面，返回时不为任何等待访问者服务，返回后再次由外向里进行扫描，即将最小柱面号紧接着最大柱面号构成循环，进行循环扫描。采用循环扫描方式后，上述请求进程的请求延迟将从原来的 $2T$ 减为 $T+S_{max}$，其中，T 为由外向里单向扫描完要访问的柱面所需的寻道时间，而 S_{max} 是将磁头从最里面被访问的柱面直接移到最外面欲访问的柱面的寻道时间。

对于上面的例子，采用循环扫描调度算法进行调度，访问的柱面次序为 65，67，98，122，124，183，14，37。除了移动臂由里向外返回(从 183 返回到 14 需移动 169 个柱面)所用的时间外，读/写磁头还需移动 153 个柱面的距离，共移动 153+169=322 个柱面。

除了先来先服务调度算法外，其余 3 种调度算法都是根据欲访问的柱面位置来进行调度的。在调度过程中可能有新的请求访问者加入，这些新的请求访问者加入时，如果读/写磁头已经超过了它们所要访问的柱面位置，则只能在以后的调度中被选择执行。

5）NStepSCAN 和 FSCAN 调度算法

(1）NStepSCAN 算法

在 SSTF、SCAN 及 CSCAN 调度算法中，都可能会出现移动臂停留在某处不动的情况，例如，有一个或几个进程对某一柱面有较高的访问频率，即这个(或这些)进程反复请求对某一柱面的 I/O 操作，从而垄断了整个磁盘设备。这一现象称为柱面臂粘着(armstickiness)。在高密度磁盘上容易出现此情况。NStepSCAN 算法是将磁盘请求队列分成若干个长度为 N 的子队列，磁盘调度将按 FCFS 算法依次处理这些子队列。而每处理一个队列时又是按 SCAN 算法，对一个队列处理完后，再处理其他队列。当正在处理某子队列时，如果又出现新的磁盘 I/O 请求，便将新请求进程放入其他队列，这样就可避免出现粘着现象。当 N 值取得很大时，会使 NStepSCAN 算法的性能接近于 SCAN 算法的性能；当 $N=1$ 时，1-StepSCAN 算法便蜕化为 FCFS 算法。

(2）FSCAN 算法

FSCAN 算法实质上是 NStepSCAN 算法的简化，即 FSCAN 只将磁盘请求队列分成两个子队列：一个是由当前所有请求磁盘 I/O 的进程形成的队列，由磁盘调度按 SCAN 算法进行处理；在扫描期间，将新出现的所有请求磁盘 I/O 的进程，放入另一个等待处理的请求队列。这样，所有的新请求都将被推迟到下一次扫描时处理。

在多道程序设计系统中，在等待访问磁盘的若干请求访问者中，有些请求访问者可能要求访问的柱面号相同，但各自要求访问同一柱面上的不同磁道，或访问同一柱面同一磁道上的不同扇区。所以，在进行移臂调度时，按照某种算法把移动臂定位到某个柱面后，应让等待访问这个柱面的各个访问者的 I/O 操作都完成后再改变移动臂的位置。

2. 旋转调度

当移动臂定位后，有多个访问者等待访问该柱面时，应怎样决定这些等待访问者的执行次序？以减少 I/O 操作总时间为目标来考虑，显然应该优先选择延迟时间最短的访问者去执行。根据延迟时间来决定执行次序的调度称为旋转调度。

进行旋转调度时应分析下列情况：

(1) 若干等待访问者请求访问同一磁道上的不同扇区。

(2) 若干等待访问者请求访问不同磁道上的不同扇区。

(3) 若干等待访问者请求访问不同磁道上的相同扇区。

对于前两种情况，旋转调度总是让首先到达读/写磁头位置下的扇区先进行传送操作。对于第3种情况，这些扇区同时到达读/写磁头位置下，旋转调度可任意选择一个读/写头进行传送操作。例如，有4个访问5号柱面的请求访问者，它们的访问要求如表7.1所示。

表7.1 访问者的访问请求

请求次序	柱面号	磁头号	扇区号	请求次序	柱面号	磁头号	扇区号
①	5	4	1	③	5	4	5
②	5	1	5	④	5	2	8

对它们进行旋转调度后，执行次序可能是①、②、④、③或①、③、④、②。

可见，当一次移动臂调度把移动臂定位到某一柱面后，还可能要进行多次旋转调度。

例7.1 假设计算机系统采用CSCAN(循环扫描)磁盘调度策略，使用2KB的内存空间记录16 384个磁盘块的空闲状态。

(1) 请说明在上述条件下如何进行磁盘块的空闲状态管理。

(2) 设某单面磁盘的旋转速度为每分钟6000转，每个磁道有100个扇区，和相邻磁道的平均移动时间为1ms。若在某时刻，磁头位于100号磁道处，并沿着磁道号增大的方向移动(如图7.4所示)，磁道号的请求队列为50、90、30、120，对请求队列中的每个磁道需要读取1个随机分布的扇区，则读完这些扇区点共需要多少时间？要求给出计算过程。

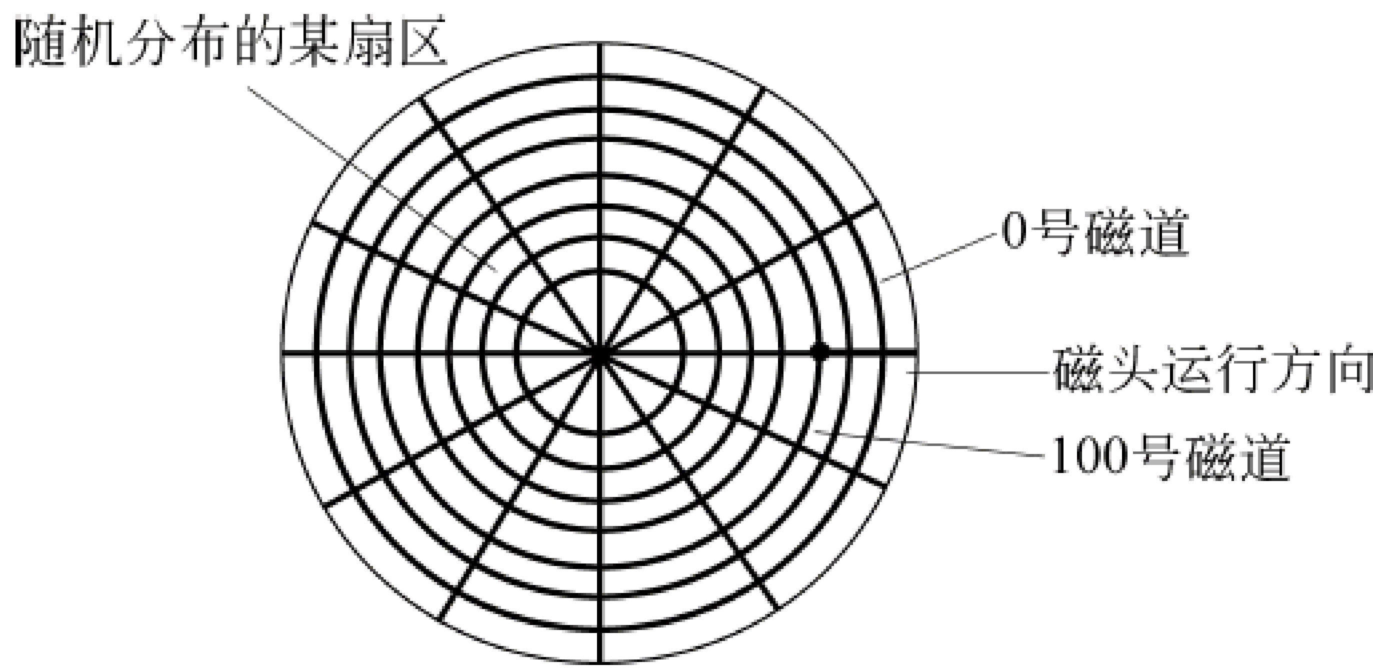


图7.4 一个磁盘的结构

(3) 如果将磁盘替代为随机访问的Flash半导体存储器(如U盘、SSD等)，是否有比CSCAN更高效的磁盘调度策略？若有，给出磁盘调度策略的名称并说明理由；若无，也说明理由。

解：

(1) 使用位示图法表示磁盘的空闲状态，每位表示一个磁盘块是否为空，共需要16384/32＝512个字＝512×4字节＝2KB，正好可以放在系统提供的内存中。

(2) 总的访问时间＝寻道时间＋延迟时间＋传输时间。

计算寻道时间：采用CSCAN调度算法访问磁道的顺序为120、30、50、90，则移动磁道长度为20＋90＋20＋40＝170，移动磁道需要总的时间为170×1＝170ms。

计算延迟时间：每分钟6000转，则平均旋转延迟时间为60×1000/(6000×2)=5ms，总的旋转延迟时间为5×4=20ms。

计算传输时间：每分钟6000转，则读取一个磁道上一个扇区的平均读取时间为10/100=0.1ms，其中的读取时间为0.1×4=0.4ms。

读取上述磁道上的所有扇区所需总时间为170+20+0.4=190.4ms。

(3) Flash半导体存储器属于可改写ROM，是一种长寿命的非易失性(在断电情况下仍能保持存储的数据信息)的存储器，数据删除不是以单个字节为单位，而是以固定的区块为单位，区块大小一般为256KB～20MB。采用FCFS(先来先服务)调度策略有更高的效率，因为Flash半导体存储器的物理结构不需要考虑寻道时间和旋转延迟时间，可以直接按I/O请求的先后顺序服务。

3. 信息的优化分布

信息在磁道上的排列方式也会影响I/O操作的时间。例如，某系统对磁盘初始化时把每一盘面分成8个扇区，今有8条逻辑记录被存放在同一磁道上供处理程序使用，处理程序要求顺序处理这8条记录，每次请求从磁盘上读一条记录，然后对读出的记录要花5ms的时间进行处理，以后再读下一条记录进行处理，直至8条记录都处理结束。假定磁盘转速为20ms/转，现把这8条逻辑记录依次存放在磁道上，如图7.5(a)所示。

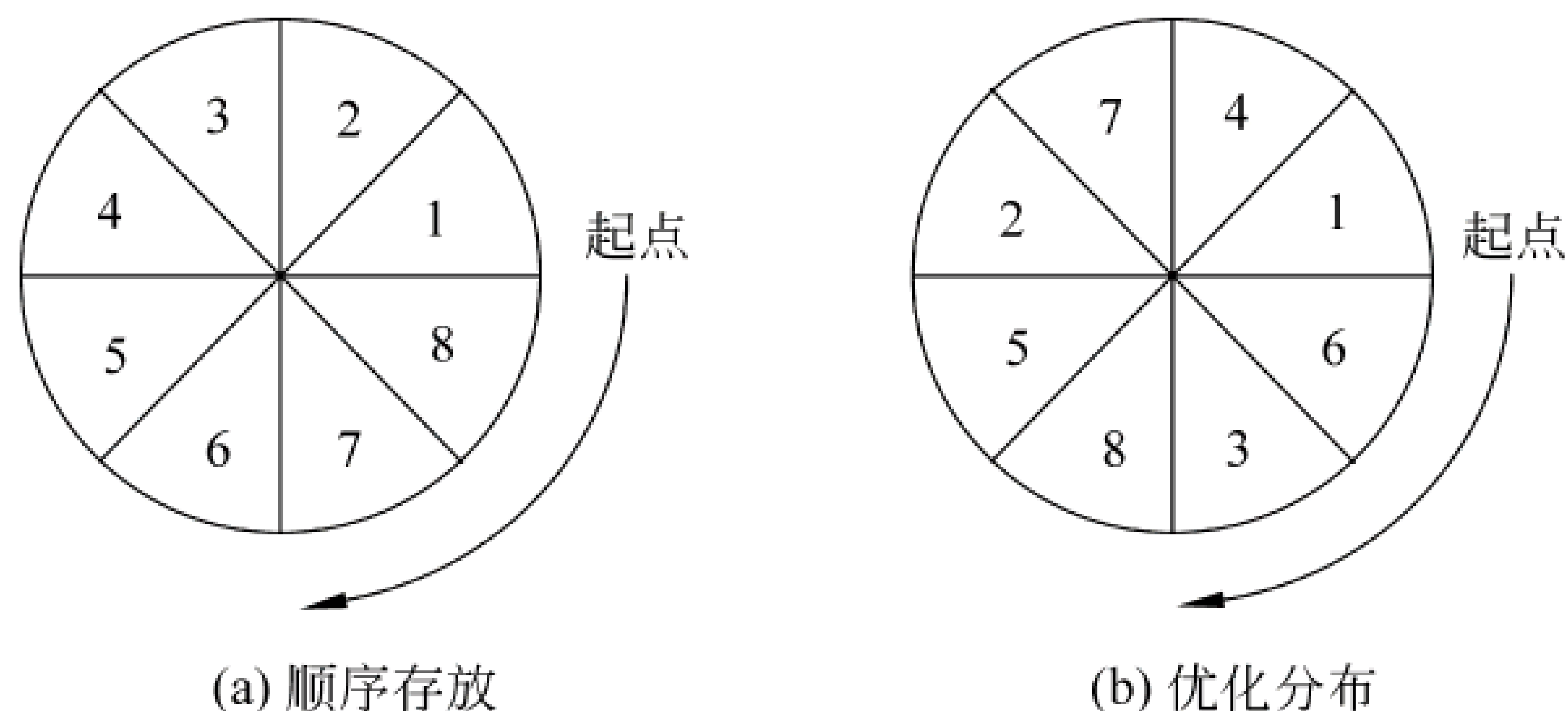


图7.5 记录的分布

显然，读一条记录要花费2.5ms的时间。当花了2.5ms的时间读出第1条记录并花5ms时间进行处理后，读/写磁头已经在第4条记录的位置，为了顺序处理第2条记录，必须等待磁盘把第2条记录旋转到读/写磁头位置下面，即要有15ms的延迟时间。于是，处理这8条记录所要花费的时间为8×(2.5+5)+7×15=165ms。

把这8条逻辑记录在磁道上的位置重新安排一下，图7.5(b)是这8条逻辑记录的最优分布。当读出一条记录并处理后，读/写磁头正好位于顺序的下一条记录位置，可立即读出该记录，不必花费等待延迟时间。于是，按图7.5(b)的安排，处理这8条记录所要花费的时间为8×(2.5+5)=60ms。可见记录的优化分布有利于减少延迟时间，从而缩短I/O操作的时间。所以，对于一些能预知处理要求的信息采用优化分布可以提高系统的效率。

7.3 中断技术

在外部设备和内存之间常用的4种数据传送方式中，除了程序直接控制方式之外，无论是中断控制方式、DMA方式还是通道控制方式，都不得不需要在设备和CPU之间进行通

信,由设备向CPU发送中断信号之后,CPU接收相应的中断信号进行处理。这几种方式的区别是中断处理的次数、数据传送方式以及控制指令的执行方式等。在计算机系统中,除了上述I/O中断之外,还存着许多其他的突发事件,例如电源掉电、程序出错等,这些也会发出中断信号通知CPU做相应的处理。因此,中断在设备管理中占有重要的地位。

7.3.1 中断及其基本概念

中断(interrupt)是指计算机在执行期间,系统内发生非寻常的急需处理事件,使得CPU暂时停止当前正在执行的程序而转去执行相应的事件处理程序,待处理完毕后又返回原来被停止处继续执行或执行优先级高的新的进程的过程。引起中断发生的事件称为中断源。中断源向CPU发出的请求中断处理信号称为中断请求,而CPU收到中断请求后转到相应的事件处理程序称为中断响应。

在有些情况下,尽管中断源发出了中断请求,但CPU内部的处理机状态字(PSW)的中断允许位已被清除,从而不允许CPU响应中断。这种情况称为禁止中断。CPU禁止中断后只有等到PSW的中断允许位被重新设置后才能接收中断。禁止中断也称为关中断,PSW的中断允许位的设置也被称为开中断。中断请求、关中断和开中断等都是由硬件实现的。

开中断和关中断是为了保证某些程序执行的原子性。

中断屏蔽是指在中断请求产生后,系统用软件方式有选择地封锁部分中断而允许其余部分的中断仍能得到响应。

中断屏蔽是通过每一类中断源设置一个中断屏蔽触发器来屏蔽它们的中断请求而实现的。不过,有些中断请求是不能屏蔽甚至不能禁止的,也就是说,这些中断具有最高的优先级。不管CPU是否是关中断的,只要这些中断请求一旦提出,CPU必须立即响应。例如,电源掉电事件所引起的中断就是不可禁止和屏蔽的中断。

7.3.2 中断处理过程

一旦CPU响应中断,转入中断处理程序,系统就开始进行中断处理。中断处理过程如下:

(1) CPU检查响应中断的条件是否满足,CPU响应中断的条件是:有来自中断源的中断请求,CPU允许中断。如果中断响应条件不满足,则中断处理无法进行。

(2) 如果CPU响应中断,则CPU关中断,使其进入不可再次响应中断状态。

(3) 保存被中断进程现场。为了在中断处理结束后能使进程正确地使返回到中断点,系统必须保存当前处理机状态字(PSW)和程序计数器(PC)等的值。这些值一般保存在特定堆栈或硬件寄存器中。

(4) 分析中断原因,调用中断处理子程序。在多个中断请求同时发生时,处理优先级最高的中断源发出的中断请求。

(5) 执行中断处理子程序。不同的中断事件的中断处理子程序不同。对陷阱来说,在有些系统中则是通过陷阱指令向当前执行进程发软中断信号后调用对应的处理子程序执行。

(6) 退出中断,恢复被中断进程的现场或调度新进程去占据处理机。

（7）开中断，CPU 继续执行。

7.3.3 中断优先级与多重中断

1. 中断的分类及中断优先级

根据系统对中断处理的需要，操作系统一般对中断进行分类，并对不同的中断赋予不同的处理优先级，以便在不同的中断同时发生时，按轻重缓急（优先级）进行处理。

根据中断产生的条件，可把中断分为外中断和内中断。

外中断是指来自处理机和内存以外的中断，包括 I/O 设备发出的 I/O 中断，外部信号中断（例如用户按 Esc 键）、各种定时器引起的时钟中断以及程序中设置的断点等引起的调试中断等。外中断在狭义上一般被称为中断。

内中断主要指在处理机和内存内部产生的中断。内中断一般称为陷阱（trap）。它包括程序运算引起的各种错误，如地址非法、校验错、页面失效、存取访问控制错、算术操作溢出、数据格式非法、除数为零、非法指令、用户程序执行特权指令（也称自愿性中断）、分时系统中的时间片中断以及从用户态到核心态的切换等都是陷阱的例子。

各中断源的优先级在系统设计时给定，在系统运行时是固定的。而陷阱的优先级则根据执行情况由系统程序动态设定。

除了在优先级的设置方面有区别之外，中断和陷阱还有如下区别：

（1）陷阱通常是由处理机正在执行的现行指令所引起的，而中断则是由与现行指令无关的中断源引起的。

（2）陷阱的处理程序提供的服务为当前进程所用，而中断处理程序提供的服务则不是为了当前进程的。

（3）CPU 在执行完一条指令之后，下一条指令开始之前响应中断，而在一条指令执行中也可以响应陷阱。

上述中断和陷阱都可以看作是硬中断，因为这些中断和陷阱要通过硬件产生相应的中断请求。而软中断则不然，它是通信进程之间用来模拟硬中断的一种信号通信方式。软中断与硬中断相同的地方是：其中断源发中断请求或软中断信号后，CPU 或接收进程在适当的时机自动进行中断处理或完成软中断信号所对应的功能。这里用“适当的时机”几个字是表示接收软中断信号的进程不一定正好在接收时占有处理机，而相应的处理机必须等到接收进程得到处理机之后才能进行。如果该接收进程是占据处理机的，那么，与中断处理相同，该接收进程在接收到软中断信号后将立即转去执行该软中断信号所对应的功能。

2. 多重中断

系统既然按中断源的优先级来对中断请求做出响应，那么肯定会出现在执行某一中断处理程序时又有更高一级的中断请求出现的情况。这就需要暂停当前的中断处理程序，转去进行新的中断处理。这种重叠中断的现象称为多重中断，又称为中断嵌套。一般情况下，在处理某级中的某个中断时，与它同级的或比它低级的中断请求均被屏蔽，不予响应，只有在处理完后再去响应和处理它们；而比它优先级高的中断请求却能中断它的处理。也就是说，当 CPU 正在执行某中断处理程序期间，又有更高优先级的中断请求发生，且 CPU 处于开中断状态时，CPU 就会暂停对原中断处理程序的执行，转去执行新的中断请求的处理程序，处理完后再返回原中断处理程序的执行或响应更高级中断。不同的计算机允许中断嵌

套的级数不同，一般为 3 级，同时提供中断栈处理技术，以保证程序从内重中断处理完成后，能正确返回外重中断继续执行。

7.4 通道技术

中断可以使设备与设备、设备与 CPU 之间实现并行处理，但并行程度不高，为了提高并行度引入了通道。

7.4.1 通道的引入

1. 通道的概念及其功能

虽然中断的引入改善了 CPU 的利用率，基本具有使 I/O 设备和处理机能更好地并行工作的条件。但是 I/O 操作毕竟还是直接由 CPU 控制的。例如，对于字符设备而言，CPU 发出启动命令后设备开始传送一个字符，CPU 利用这个短暂的空闲时间转去执行别的程序。这时，外设与 CPU 是并行的。一旦一个字符传送完毕，外设就要发生中断请求，要求 CPU 脱出身来干预自己的 I/O 工作。为把 CPU 从这种繁忙的事务中解脱出来，使 I/O 设备管理不再依赖 CPU，通道技术应运而生。通道是一种硬件设施，本质上是一台专门用来管理 I/O 的处理器，它根据来自 CPU 的 I/O 操作指令，以独立于 CPU 的方式执行有关的通道程序，承担起 I/O 操作的组织、管理、数据传送以及结束处理等工作。只有在结束了 CPU 委托的 I/O 任务之后才向 CPU 发出中断请求信号。通道的出现使 CPU 基本上摆脱了 I/O 的处理事务，充分保证了 CPU 与外设操作的并行进行，从而使整个计算机系统的效率有可能大大提高。

具有通道的计算机一般是大、中型、数据通量很大的计算机。

通道的基本功能是执行通道指令、组织外部设备和内存进行数据传输，按 I/O 指令要求启动外部设备，向 CPU 发出中断等，具体有以下 5 项功能：

（1）接受 CPU 的 I/O 指令，按指令要求与指定的外部设备进行通信。

（2）从内存中取出属于该通道程序的指令，经译码后向设备控制器和设备发出各种命令。

（3）组织外部设备和内存之间进行数据传输，并根据需要提供数据缓冲的空间，以及提供数据存入内存的地址和传送的数据量。

（4）从外部设备得到设备的状态信息，形成并保存通道本身的状态信息，根据要求将这些状态送到内存的指定单元，供 CPU 使用。

（5）将外部设备的中断请求和通道本身的中断请求按次序及时报告 CPU。

2. 通道的连接

系统引入了通道之后，整个 I/O 系统结构呈现如图 7.6 所示的四级连接、三级控制形式。从图中可以看出，一个通道可连接若干台控制器，一台控制器可连接多台外部设备。CPU 执行 I/O 指令，对通道实施控制；通道执行通道命令对控制器实施控制；控制器发出动作序列对设备实施控制；设备则执行相应的 I/O 操作，该操作具体实施信息的 I/O。

通道、控制器和设备之间的连接还可以采用图 7.7 所示的多通路连接方案。使每台设备与主机的连接有多条道路。

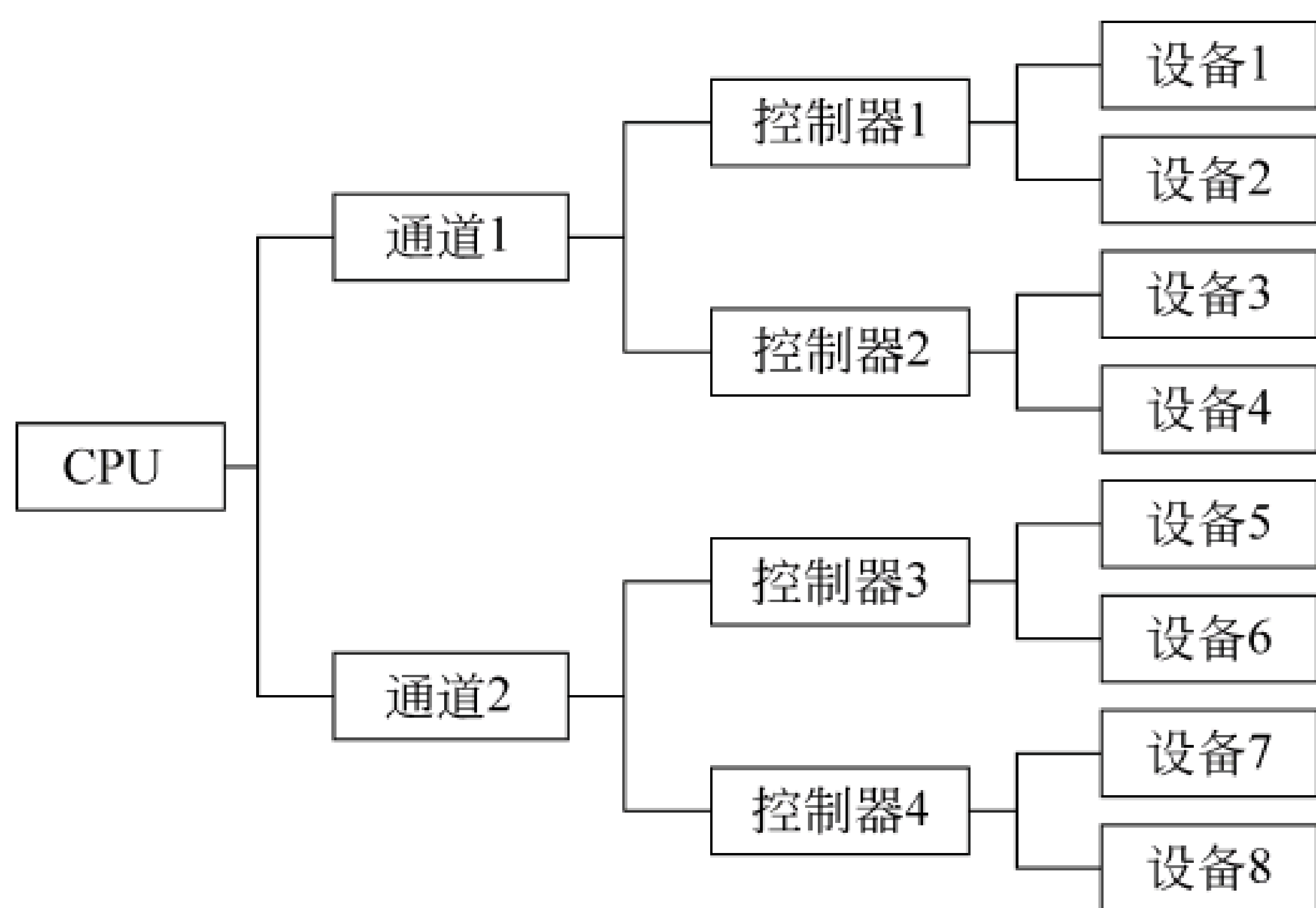


图 7.6 I/O 系统的四级结构

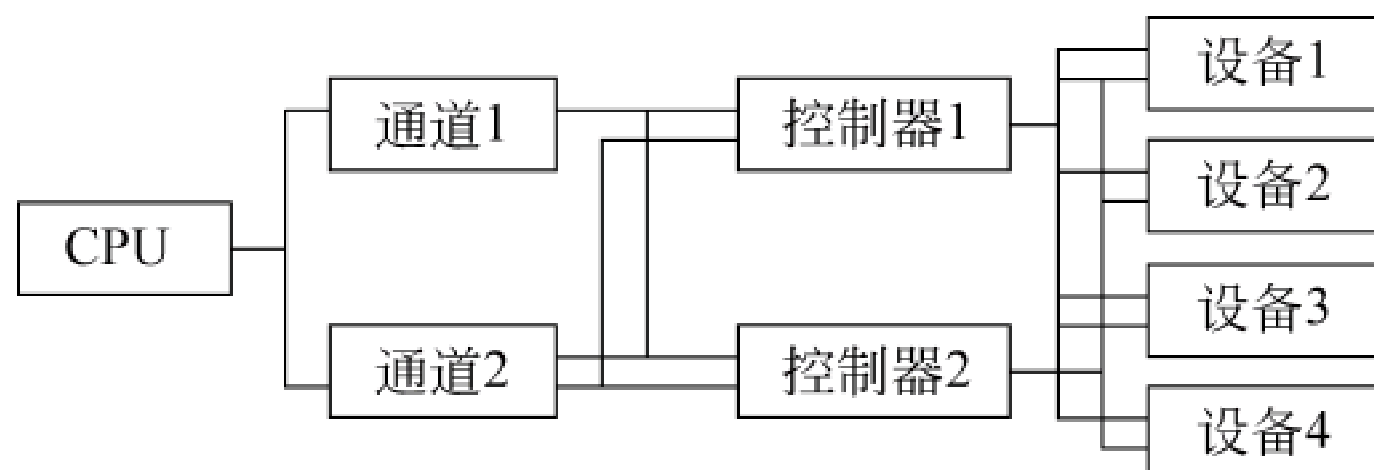


图 7.7 I/O 系统的多通路连接

将图 7.6 和图 7.7 两种连接方式做一比较，不难看出采用图 7.6 的多通路连接方式，4 台设备中的任意两台都可同时启动，效率高，具有灵活性。同时，由于每台设备均有 4 条通路与主机相连，所以这种连接的可靠性好。即使某条通路出现故障，也可通过其他通路继续访问，不会严重影响数据的传输。

CPU 是执行 I/O 指令以及处理来自通道的中断，实现对通道的管理，来自通道的中断有两种：一种是数据传送结束中断，另一种是故障中断。

通道通过使用通道指令控制设备控制器进行数据传输操作，并以通道状态字接收设备控制器反映的外部设备的状态。因此设备控制器是通过通道对 I/O 设备实现传输控制的机构。

7.4.2 通道类型

按信息交换的方式和连接设备的不同，通道可分为 4 类。

1. 字节多路通道

字节多路通道是连接大量慢速或中速外部设备的，如软盘输入输出机、纸带输入输出机、卡片输入输出机、控制台打字机等设置。按字节交叉方式工作，即为一台设备传送一个字节后，便立即腾出字节多路通道转去为另一台设备传送一个字节。由于字节通道的速度较高，最大信息流通量可达几十 KB/s，而所连接的主要是低速设备，因此完全能以这种交叉方式来同时为多台慢速设备服务。

2. 选择通道

选择通道的数据传送是按成组方式进行的，即每次传送一批数据，故传送速度很高。它主要用于连接磁带、磁鼓和磁盘等高速外存储设备。选择通道的特点是以 CPU 委托的 I/O

操作作为一次服务的对象,所以它在一段时间内只能为一个设备工作,完成了这一I/O请求后,再转去为另一台设备服务。

选择通道在数据传送期间只为一台设备服务,而这类设备的辅助操作时间很长,如磁盘机平均寻道时间为20～30μs,磁带机走带时间可长达几分钟,在这样长的时间里通道只能处于等待状态,因此这种方式的通道利用率是较低的。

3. 数组多路通道

数组多路通道是结合了选择通道传送效率高,字节多路通道能进行分时并行操作的优点而形成的又一种通道方式,它的基本思想是:当某设备进行数据传送时,通道只为该设备服务;当设备在执行寻址等操作时,通道暂时断开与这个设备的连接,挂起该设备的通道程序,去为其他设备服务。所以它很像一个多道程序的处理器。

数组多路通道可同时连接多台外部设备,数据传送按成组方式进行,几个通道程序分时并行工作。

由于它具有很高的数据传送速率,又可获得令人满意的通道利用率,所以被广泛地用来连接高、中速外部设备。

字节多路通道和数组多路通道的共同之处是:它们都是多路通道,在一段时间内交替执行多个通道程序,使这些设备并行工作。

字节多路通道和数组多路通道的不同之处主要有以下两点:

(1)数组多路通道允许多个设备同时工作,但只允许一个设备进行传输操作,其他设备进行控制操作;字节多路通道既允许设备同时进行控制操作,也允许它们同时进行数据传输操作。

(2)数组多路通道与设备之间进行数据传输的基本单位是数据块;字节多路通道与设备之间进行数据传输的基本单位是字节。

图7.8所示为IBM 370系统的结构示意图,它包含了字节多路通道、选择通道和数组多路通道。

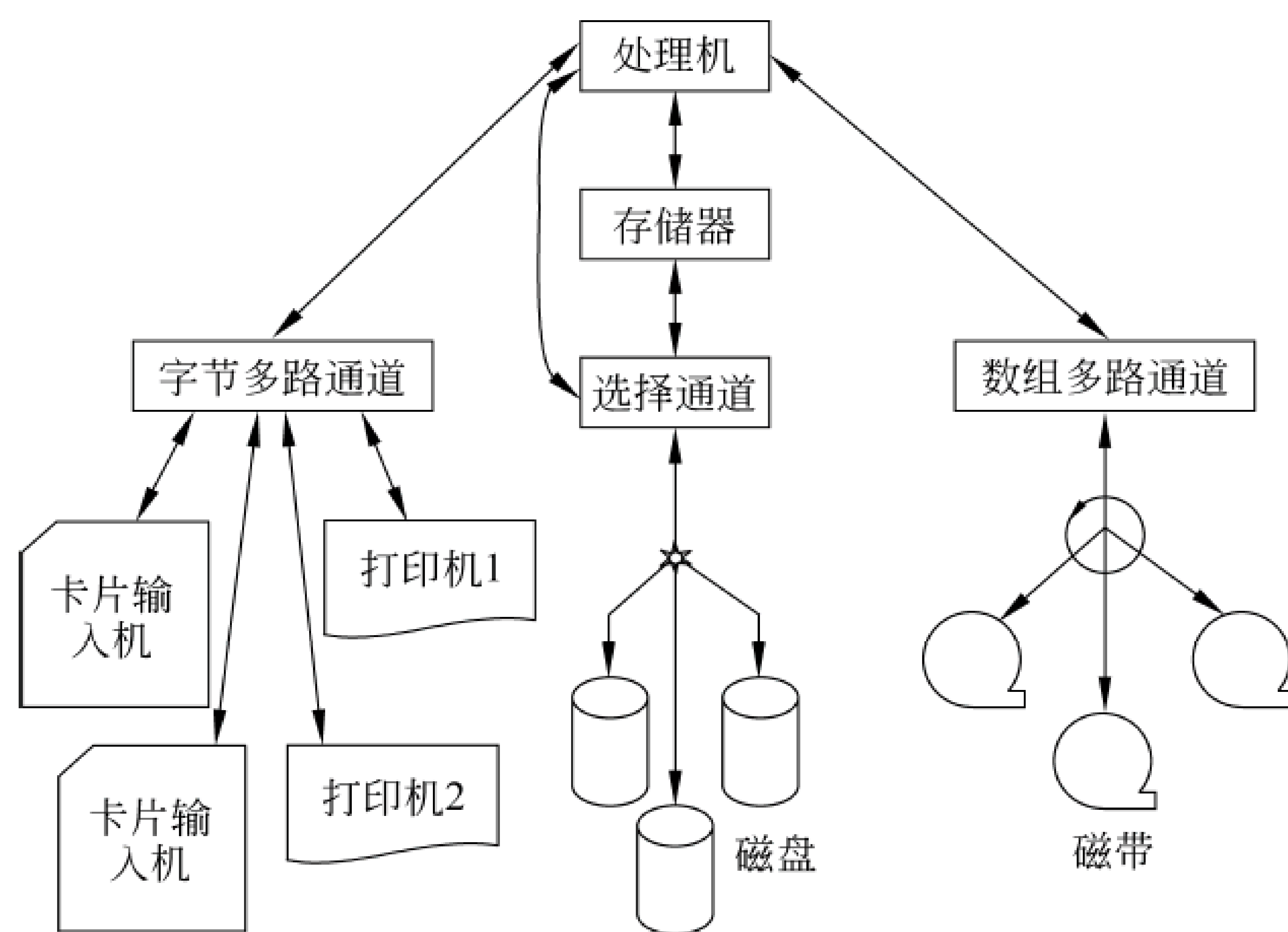


图7.8 IBM 370系统的结构示意图

4. 通道适配器

在某些系统中,还常配有通道适配器,如磁盘文件适配器和控制台适配器等。

通道适配器是将通道与某种设备控制器结合在一起的专用性质的通道。它用于将某些特定的设备相连接,这些设备一般是系统常用的或必备的。它的逻辑设计是针对专用设备,适配器和设备之间只有专用接口线。

由于通道适配器不具有通用性,是通道和设备控制器结合在一起形成的,因此它的性能价格比要比一般通道的高。又因为它所连接的设备大部分是系统必用的,所以它的专用性并不影响它的广泛使用。

7.4.3 通道指令和通道程序

1. 通道指令

由于通道是专门用于I/O的处理器,因此它具有与中央处理器类似的组成。首先,通道有自己的指令系统,被称为通道指令或通道命令字(Channel Command Word,CCW),一般有如下3种基本命令:

(1) 数据传输。包括读、反读、写和断定(读设备状态)。

(2) 设备控制。包括换页和磁带反绕等。

(3) 转移。通道程序内部的控制转移。

通道命令由操作码、数据地址、特征位和交换字节个数等组成,IBM系统的通道命令格式如图7.9所示。

0　　　7	31	36 37	47 48	63
操作码	数据地址	特征位	不用	传送字节个数

图7.9 IBM系统的通道命令格式

(1) 操作码:表示要执行的命令。

(2) 数据地址:规定了本命令所访问的数据区的起始地址。

(3) 特征位:进一步限定本命令的含义,给出连接方式或命令特点。

(4) 传送字节个数:规定了数据区的字节数。数据地址和传送字节个数主要用于数据传输型命令。

2. 通道程序

中央处理器启动通道工作时,应把要求通道"做什么和怎么做"告诉通道。因此操作系统必须按用户的要求和设备的特性来规定通道的工作。为了使操作系统能用同样的手段使用种类繁多、特性各异的外部设备,计算机硬件提供通道指令,操作系统用一组通道指令来规定通道执行一次I/O操作应做的工作,这一组通道指令就组成了一个通道程序。通道被启动后,就依次执行预定的通道程序中的一条条通道指令,从而实现对外部设备的操作控制。

要启动外部设备按指定的要求工作,首先要把这个指定的要求用通道程序表示出来。在编制通道程序时应根据系统提供的通道命令格式和通道命令码来进行。下面用实例来说明通道程序的设计。

例7.2 用户把自己组织好的一行信息已经存放在主存储器 L 单元开始的区域中,该

行信息为"Operating System",连空格在内共16个字符。用户要求把该行信息从打印机打印输出,要求打印在新的一页第4行位置。

为了使打印机能按要求打印,可组织一个含有3条通道命令的通道程序,把组织好的通道程序存放在主存储器 K 单元开始的区域中,设计的通道程序如表7.2所示。

表7.2 例7.2中的通道程序

主存地址	命　令　码	数据主存地址	标　志　码	传送字节个数
K	07	000000	60	0001
$K+8$	EF	000000	60	0001
$K+16$	F9	L	00	0010

其中命令码07表示"对折页线"操作(即走纸到新的一页开始),命令码EF表示"走纸3行"的操作(即使打印针在第4行位置),命令码F9表示"打印一行信息"的操作。这两条通道命令是设备控制类命令,传送字节个数应填一个非"0"数,在这里填了"1",标志码是非"0"字符,表示有后继的通道命令,这两条命令不需要进行数据传输,也没有控制信息,因此,数据主存地址为"0",表示该项省略。最后一条通道命令指出了把 L 单元开始的信息打印输出,共16个字符(16的二进制表示为10),标志码为"0",表示控制打印机执行的操作到此结束。

例7.3 用户要求把磁带上的第一块(块长为512B)信息读入到主存储器 A 单元开始的区域。

为了使磁带机能读出第一块信息,需组织一个含有两条通道命令的通道程序,把组织好的通道程序存放在 T 单元开始的区域中,通道程序设计如表7.3所示。

表7.3 例7.3中的通道程序

主存地址	命　令　码	数据主存地址	标　志　码	传送字节个数
T	07	000000	60	0001
$T+8$	02	A	60	0100

对磁带机来说,命令码07表示磁带"反绕到始点"的操作,命令码02表示"读"操作。第一条通道命令控制磁带返回始点,第二条通道命令把磁带上一块长度为512B(512的二进制表示为0100)的信息读入主存储器 A 单元开始的区域。

7.4.4 通道的工作过程

1. 通道地址字

编制好的通道程序是存放在主存中的,为了使通道能取到通道命令并执行,必须把存放通道程序的主存起始地址告诉通道。在具有通道的计算机系统中,在主存中设置一个固定单元用来存放当前启动外部设备时要求通道执行的通道程序的首地址。这个用来存放通道程序首地址的主存固定单元称为通道地址字(Channel Address Word,CAW)。通道被启动后从CAW中取内容即第一条通道命令(CCW)的地址送入CAWR(通道地址寄存器)中记录下来,通道按指示的地址取通道的第一条指令送通道指令寄存器(CCWR)中,并修改

CAWR 的内容使其指向下一条通道指令，以后就可从 CAWR 指示的内存单元取通道命令送入 CCWR 中，解释执行。可见 CAWR 相当于 CPU 的程序计数器，CCWR 相当于 CPU 的指令寄存器。

2. 通道状态字

当通道被启动成功后，要控制指定的设备完成通道命令规定的操作，通道在执行通道程序时把通道和设备执行操作的情况随时记录下来，汇集在一个通道状态字(Channel Status Word，CSW)中，IBM 系统中的通道状态字也采用双机器字表示，格式如图 7.10 所示。

0　　7	8　　31	32　　39	40　　47	48　　63
	通道命令地址	设备状态	通道状态	剩余字节个数

图 7.10　IBM 系统中的通道状态字

其中各字段的含义如下：

(1) 通道命令地址。通道从 CAW 中取到通道程序中的第一条命令的存放地址，并把该地址存入 CSW 中。以后 CSW 中的通道命令地址随 CAWR 内容的改变而改变，即通道命令地址指向后继通道命令的地址。

(2) 设备状态。把设备控制器和设备能识别到的情况记录在“设备状态”字段中。例如忙、控制器结束、通道结束、设备出错和设备特殊等状态。

(3) 通道状态。通道识别到的情况，例如接口错、控制错、数据错和通道程序错等，都记录在“通道状态”字段中。

(4) 剩余字节个数。指最后一次执行的那条通道命令在操作结束后还剩余多少字节未传输。

3. 通道的工作过程

每一条通道命令规定了设备的一种操作，若干条通道命令按照一定的方式组织起来，构成了通道程序，它限定了设备所应执行的各种操作及顺序。

通道与 CPU 共用一个主存，通道程序是存放在主存里的，其起始地址被送入通道地址字寄存器(CAWR)。CPU 启动 I/O 指令，通道接收后，就按如下步骤工作：

(1) 根据 CAWR 中的地址到内存取一条通道命令存入通道控制字寄存器(CCWR)中，同时修改 CAWR 中的内容，以指向下一条通道命令。

(2) 通道对 CCWR 中的命令进行解释并执行，进行一个实际的 I/O 操作。

(3) 转向(1)，继续执行下一条通道命令，直至通道程序结束运行，转(4)。

(4) 发出中断信号，通知 CPU 本次 I/O 任务已经完成。

7.5 缓冲技术

中断和通道提高了设备与设备、设备与 CPU 的并行处理能力，但 CPU 和外设之间的速度差异问题没有得到彻底解决，为此引入缓冲技术。

7.5.1 缓冲的引入

引入缓冲的原因主要有 3 个：

(1) 为了匹配外设与CPU之间的处理速度。虽然中断、DMA和通道技术使得系统中设备和设备、设备和CPU等得以并行工作,但是,外部设备和CPU的处理速度不匹配的问题是客观存在的。这限制了和处理机连接的外设台数,且在中断方式时造成数据丢失。从而,外部设备和CPU处理速度不匹配的问题极大地制约了计算机系统性能的进一步提高,同时也限制了系统的应用范围。

(2) 为了减少中断次数和中断处理时间。从减少中断的次数看,存在着引入缓冲区的必要性。在中断方式时,如果在I/O控制器中增加一个容量为100个字符的缓冲器,则I/O控制器对处理机的中断次数将降低100倍,即等到能存放100个字符的字符缓冲区装满之后才向处理机发一次中断。这将大大减少处理机的中断处理时间。

(3) 为了解决DMA或通道方式时的瓶颈问题。即使是使用DMA方式或通道方式控制数据传送时,如果不划分专用的内存区或专用缓冲器来存放数据的话,也会因为要求数据的进程所拥有的内存区不够或存放数据的内存始址计算困难等原因而造成某个进程长期占有通道或DMA控制器及设备,从而产生瓶颈问题。在设备管理中引入了用来暂存数据的缓冲技术,以解决DMA或通道方式时的瓶颈问题。

根据I/O控制方式,缓冲的实现方法有两种:一是采用专用硬件缓冲器,例如I/O控制器中的数据缓冲寄存器;另一方法是在内存划出一个具有n个单元的专用缓冲区,以便存放I/O的数据。内存缓冲区又称软件缓冲。

7.5.2 缓冲的种类

根据系统设置的缓冲器个数,可把缓冲技术分为单缓冲、双缓冲、多缓冲和缓冲池4种。

单缓冲是在设备和处理机之间设置一个缓冲器。设备和处理机交换数据时,先把被交换数据写入缓冲器,然后,需要数据的设备或处理机从缓冲器取走数据。尽管单缓冲能匹配设备和处理机的处理速度,但是,设备和设备之间不能通过单缓冲达到并行操作。

双缓冲是解决两台外设、打印机和终端之间的并行操作问题的办法。两个缓冲器中,一个作为数据输入缓冲器(区),另一个作为数据输出缓冲器(区)。然而它不能用于实际系统中的并行操作。这是因为计算机系统中的外部设备较多,另外,双缓冲也很难匹配设备和处理机的处理速度。因此,现代计算机系统中一般使用多缓冲或缓冲池结构。

多缓冲是把多个缓冲区连接起来组成两部分,一部分专门用于输入,另一部分专门用于输出的缓冲结构。

缓冲池则是把多个缓冲区连接起来统一管理,既可用于输入又可用于输出的缓冲结构。

显然,无论是多缓冲还是缓冲池,由于缓冲区是临界资源(不可同时使用的资源),在使用缓冲区时都有一个申请、释放和互斥的问题(详见第8章)。下面以缓冲池为例,介绍缓冲的管理。

7.5.3 缓冲池的管理

1. 缓冲池的结构

缓冲池由多个缓冲区组成。一个缓冲区由两部分组成:一部分是用来标识该缓冲区和用于管理的缓冲首部,另一部分是用于存放数据的缓冲体。这两部分有一一对应的映射关系。对缓冲池的管理是通过对每一个缓冲区的缓冲首部进行操作实现的。

缓冲首部包括设备号、设备上的数据块号(块设备时)、互斥标识位以及缓冲队列连接指针和缓冲器号等。

系统把各种缓冲区按其使用状况连成3种队列,队列构成如图7.11所示。

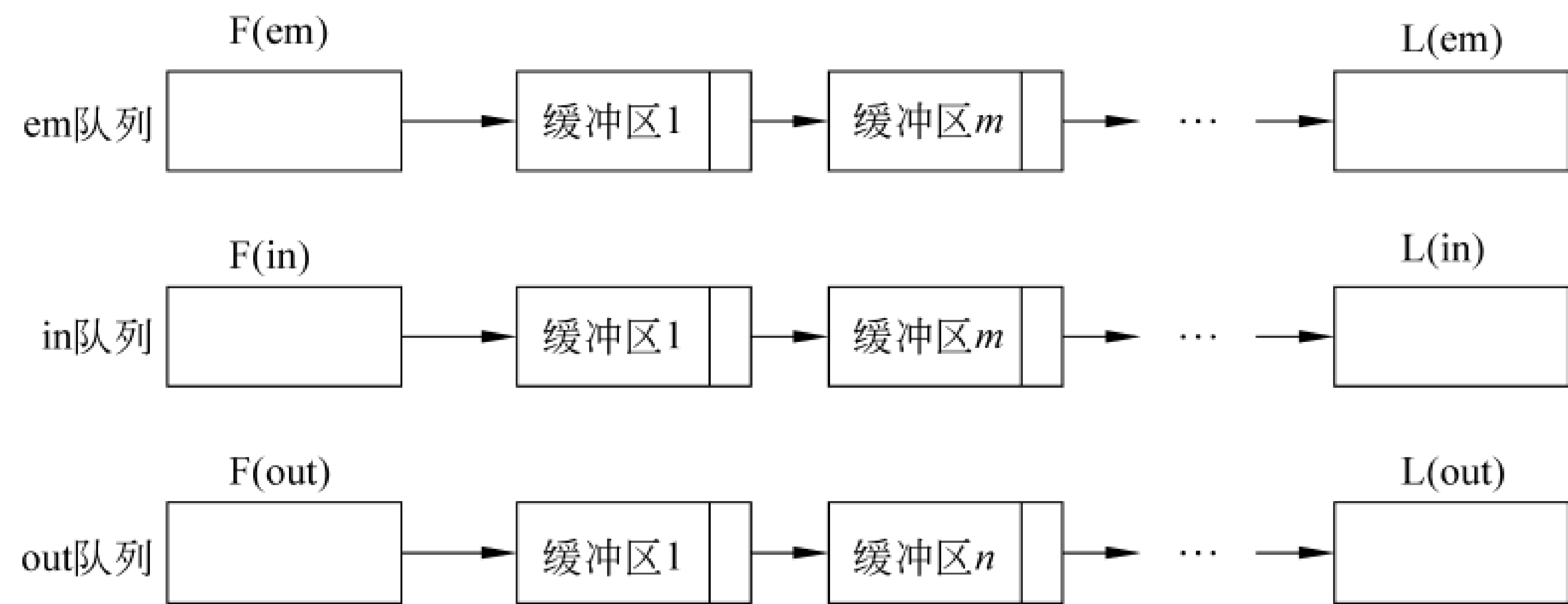


图7.11 缓冲区队列

(1) 空白缓冲队列em,其队首指针为F(em),队尾指针为L(em)。

(2) 装满输入数据的输入缓冲队列in,其队首指针为F(in),队尾指针为L(in)。

(3) 装满输出数据的输出缓冲队列out,其队首指针为F(out),队尾指针为L(out)。

建立3种缓冲队列之后,系统(或用户进程)从这3种队列中申请和取出缓冲区,并用得到的缓冲区进行存数、取数操作,在存数、取数操作结束后,再将缓冲区放入相应的队列。这些缓冲区被称为工作缓冲区。在缓冲池中,有4种工作缓冲区:

(1) 用于收容设备输入数据的收容输入缓冲区hin。

(2) 用于提取设备输入数据的提取输入缓冲区sin。

(3) 用于收容CPU输出数据的收容输出缓冲区hout。

(4) 用于提取CPU输出数据的提取输出缓冲区sout。

缓冲池的工作缓冲区如图7.12所示。

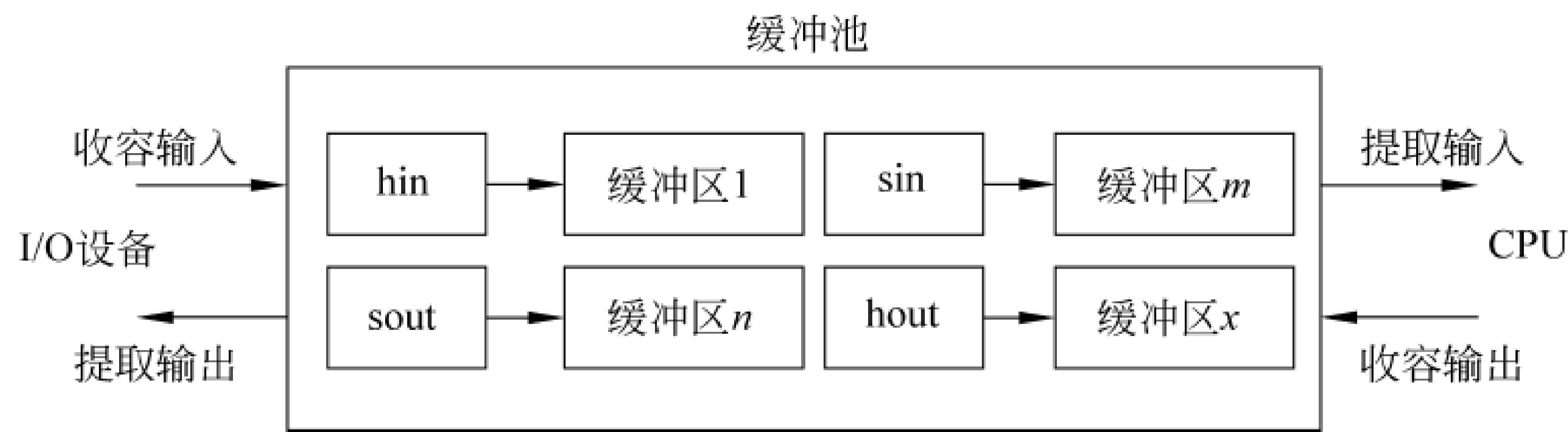


图7.12 缓冲池的工作缓冲区

2. 缓冲池的管理

对缓冲区的管理由如下几个操作组成:

(1) 从3种缓冲区队列中按一定的选取规则取出一个缓冲区的过程。

(2) 把缓冲区按一定的规则插入相应的缓冲区队列的过程。

(3) 供进程申请缓冲区用的过程,供操作(1)使用。

(4) 供进程将缓冲区放入相应缓冲区队列的过程,供操作(2)使用。

例如,从外设输入数据送到输入缓冲区,操作过程是:首先从空白缓冲区队列(em)中

按一定的选取规则取出一个空闲缓冲区,把输入的数据放入该空闲缓冲区中,然后把装满数据的缓冲区按一定的规则插入相应的输入缓冲队列(in)。

7.6 设备分配

前面已经介绍了I/O数据传送控制方式及与其紧密相关的中断技术、通道技术与缓冲技术。不过,在讨论这些问题时,已经做了如下假定:即每一个准备传送数据的进程都已申请到了它所需要的外部设备、控制器和通道。事实上,由于设备、控制器和通道资源的有限性,不是每一个进程随时随地都能得到这些资源。进程必须首先向设备管理程序提出资源申请,然后,由设备分配程序根据相应的分配算法为进程分配资源。如果申请进程得不到它所申请的资源时,将被放入资源等待队列中等待,直到所需要的资源被释放。

7.6.1 设备的独立性

1. 设备的绝对号和相对号

计算机系统中配置有各种不同的外部设备,每一类设备又可以有多台。为了对这些设备进行管理,计算机系统为每个设备给定一个编号,以便区分和识别,这个确定的编号称为设备的绝对号。

在多道程序设计系统中,因为用户无法知道哪台设备被其他用户占用了,哪台设备是空闲的,所以,一般情况下用户不直接使用设备的绝对号。用户可以向系统说明所要使用的设备类型,至于实际使用哪一台,由系统根据该类设备的分配情况来决定。有时用户可能要求同时使用几台同类设备,为了避免使用时的混乱,用户可以把自己要求使用的若干台同类设备进行编号,由用户在程序中定义的设备编号称为设备的相对号。于是,用户是按设备类型和相对号来提出使用设备的要求。系统为用户分配了具体设备后,建立绝对号与设备类型和相对号的对应关系。这样,系统根据用户程序执行时给出的使用要求就能知道实际应启动哪台设备。

2. 设备的独立性

作业或进程申请设备时,应指定需要什么设备,指定的方式可以有两种:一种是指定设备的绝对号,另一种是指定设备类型和相对号。

如果用绝对号来指定设备,系统应该把与绝对号对应的那台设备分配给作业或进程。如果指定的这台设备已经被其他作业或进程占用或者有故障时,则作业或进程的申请就得不到满足。于是,该作业或进程就暂时不能执行。

通常,申请设备时不是具体指定要哪台设备,而是提出要申请哪类设备多少台,即在用户程序中用设备类型和相对号提出使用设备的要求。这种方式使设备分配的适应性好、灵活性强,这是因为:

(1) 系统只要从指定的那一类设备中找出"好的且尚未分配的"设备来进行分配。

(2) 万一分配给用户的设备在使用中出了故障,系统可以从同类设备中找另一台"好的且尚未分配的"设备来替换。

所以,采用设备类型和相对号的方式使用设备时,用户编制程序时不必指定特定的设备。在程序中使用由设备类型和相对号定义的逻辑设备。程序执行时系统根据用户指定的

逻辑设备转换成与其对应的具体物理设备，并启动该物理设备工作。于是用户编制程序时使用的设备与实际使用哪台设备无关。把这种特性称为设备的独立性。

7.6.2 设备分配的原则

设备分配的原则是根据设备特性、用户要求和系统配置情况决定的。设备分配的总原则是既要充分发挥设备的使用效率，尽可能地让设备忙，但又要避免由于不合理的分配方法造成进程死锁（详见第8章）；另外还要做到把用户程序和具体物理设备隔离开来，即用户程序面对的是逻辑设备，而分配程序将在系统中把逻辑设备转换成物理设备之后，再根据要求的物理设备号进行分配。

设备分配方式有两种，即静态分配和动态分配。

静态分配方式是在用户作业或进程开始执行之前，由系统一次分配该作业或进程所要求的全部设备、控制器和通道。一旦分配之后，这些设备、控制器和通道就一直为该作业或进程所占用，直到该作业或进程被撤销。静态分配方式不会出现死锁，但设备的使用效率低。因此，静态分配方式并不符合设备分配的总原则。

动态分配方式在作业或进程执行过程中根据执行需要进行。当作业或进程需要设备时，通过系统调用命令向系统提出设备请求，由系统按照事先规定的策略给进程分配所需要的设备、I/O控制器和通道，一旦用完之后，便立即释放。动态分配方式有利于提高设备的利用率，但如果分配算法使用不当，则有可能造成死锁。

7.6.3 设备分配策略

与进程调度相似，动态设备分配也是基于一定的分配策略的。常用的分配策略有先请求先分配、优先级高者先分配策略等。

1. 先请求先分配

当有多个进程对某一设备提出I/O请求时，或者是在同一设备上进行多次I/O操作时，系统按提出I/O请求的先后顺序，将进程发出的I/O请求命令排成队列，其队首指向被请求设备的DCT（设备控制表）。当该设备空闲时，系统从该设备的请求队列的队首取下一个I/O请求，将设备分配给发出这个请求的进程。

2. 优先级高者先分配

优先级高者指发出I/O请求命令的进程的优先级高。这种策略和进程调度的优先数法是一致的，即进程的优先级高，它的I/O请求也优先予以满足。对于相同优先级的进程来说，则按先请求先分配策略分配。因此，优先级高者先分配策略把请求某设备的I/O请求命令按进程的优先级组成队列，从而保证在该设备空闲时，系统能从I/O请求队列队首取下一个具有高优先级的进程发来的I/O请求命令，并将设备分配给发出该命令的进程。

7.6.4 设备分配所使用的数据结构和分配算法

1. 设备分配所使用的数据结构

设备的分配和管理通过下列数据结构进行。

1) 设备控制表(Device Control Table,DCT)

DCT 反映设备的特性、设备和 I/O 控制器的连接情况。DCT 主要包括设备标识、使用状态和等待使用该设备的进程队列等。系统中的每个设备都必须有一张 DCT,且在系统生成时或在该设备和系统连接时创建,但表中的内容则根据系统执行情况而被动态地修改。

2) 系统设备表(System Device Table,SDT)

SDT 为整个系统一张,它记录已被连接到系统中的所有物理设备的情况,并为每个物理设备设一个表项。SDT 的每个表项包括的内容主要有 DCT 指针、正在使用设备的进程标识、设备类型和设备标识符等。

SDT 的主要意义在于反映系统中设备资源的状态,即系统中有多少设备,有多少是空闲的,又有多少已分配给了哪些进程。

3) 控制器控制表(COntroller Control Table,COCT)

COCT 是每个控制器一张,它反映 I/O 控制器的使用状态以及和通道的连接情况等(在 DMA 方式时,该项是没有的)。COCT 主要包括控制器标识、控制器状态、通道指针和等待队列指针等。

4) 通道控制表(CHannel Control Table,CHCT)

该表只在通道控制方式的系统中存在,每个通道一张。CHCT 主要包括通道标识符、通道忙/闲标识、等待获得该通道的进程等待队列的队首指针与队尾指针等。

SDT、DCT、COCT 及 CHCT 如图 7.13 所示。

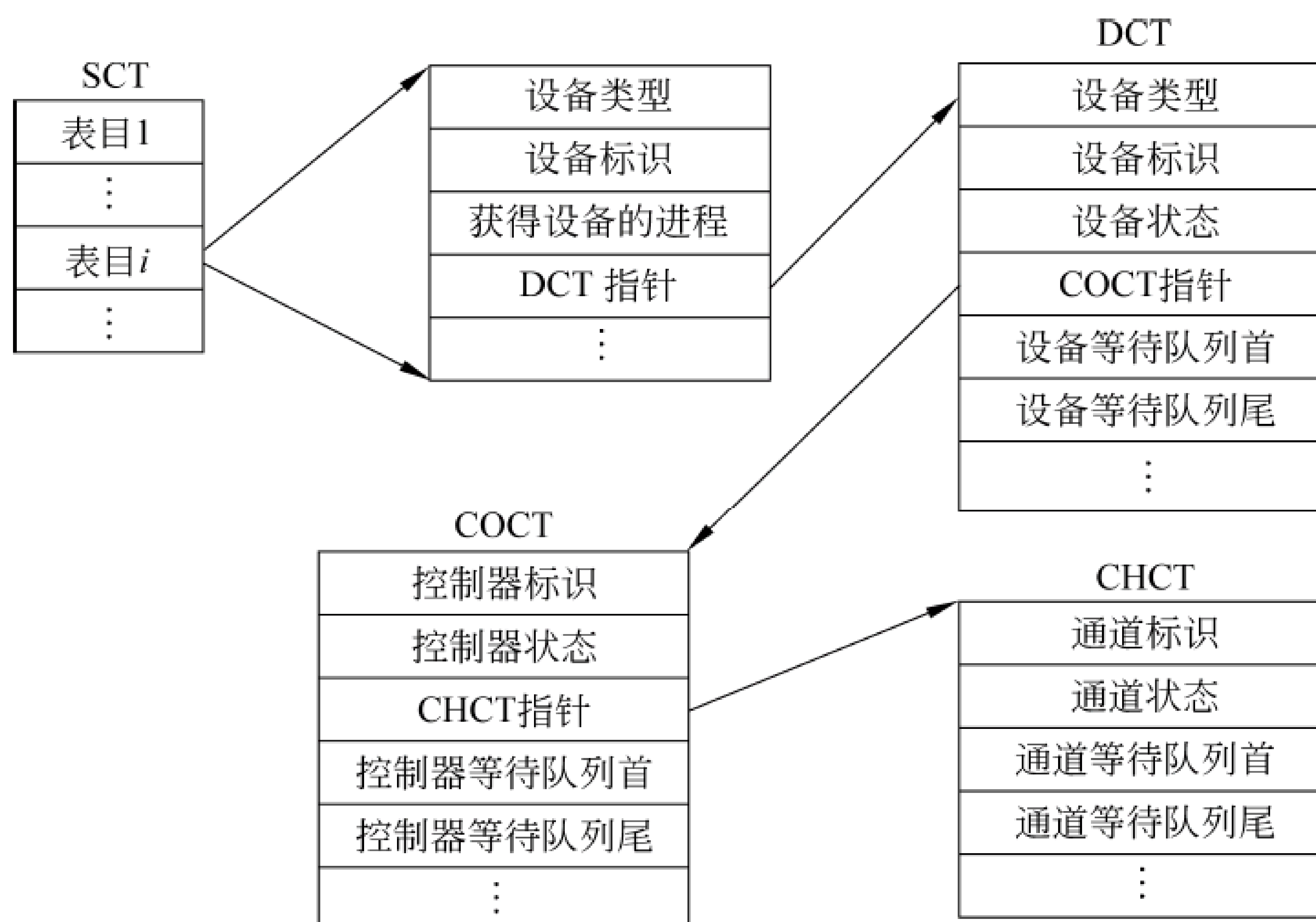


图 7.13　数据结构及其关系

显然,一个进程只有获得了通道、控制器和所需设备三者之后,才具备了进行 I/O 操作的物理条件。

2. 设备分配算法

根据设备分配策略和原则,使用系统提供的 SDT、DCT、COCT 及 CHCT 等数据结构,当某个进程提出 I/O 设备请求之后,就可按图 7.14 所示流程进行设备分配。

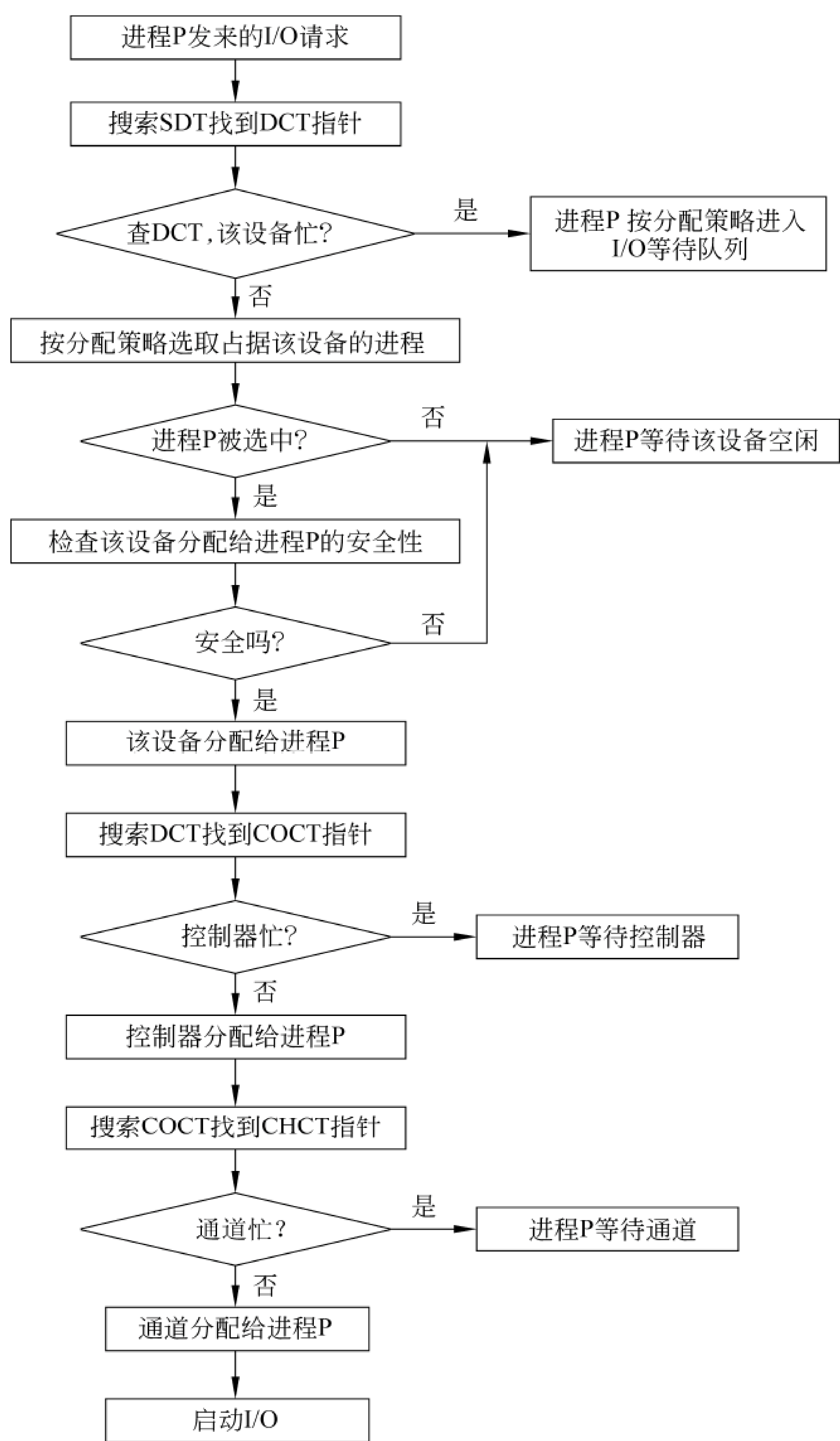


图 7.14　设备分配算法流程

7.7 虚拟设备

操作系统利用共享设备来模拟独占设备的工作，当系统只有一台输入设备和一台输出设备的情况下，可允许两个或两个以上的作业并行执行，并且每个作业都感觉到获得了供自己独占使用的输入设备和输出设备。操作系统采用的这种技术为用户提供了“虚拟设备”。

7.7.1 虚拟设备的引入

人们已经知道,像输入机、打印机等独占使用的设备采用静态分配方式,既不能充分利用设备,又不利于提高系统效率,所以现代操作系统中都提供虚拟设备来解决这些问题。用可共享的磁盘来模拟输入机和打印机等独占设备的工作,使每个作业都感到各自拥有独占使用的设备且它们的传输速度与磁盘的传输速度一样快。这种用一类物理设备模拟另一类物理设备的技术,使各作业在执行期间只使用虚拟的独占设备而不直接使用物理的独占设备,每个作业也不必独占输入机和打印机等独占设备。这种技术使独占使用的设备变成了可共享的设备,使得设备的利用率和系统效率都能得到提高。

7.7.2 虚拟设备的实现

1. 基本条件

实现虚拟设备必须要有一定的硬件和软件条件为基础。对硬件来说,必须配置大容量的磁盘,要有中断装置和通道,具有中央处理器与通道的并行工作的能力。对操作系统来说,应采用多道程序设计技术。

2. 实现原理

虚拟设备的实现原理如下。

把一批作业的全部信息通过输入设备预先传送到磁盘上。在多道程序设计的系统中,可从磁盘上选择若干个作业同时装入主存,并让它们同时执行。由于作业的信息已全部在磁盘上,故作业执行时不必启动输入机读信息,而可以从共享的磁盘上读取各自的信息。把作业产生的结果也暂时存放在磁盘上而不直接启动打印机输出,直到一个作业得到全部结果而执行结束时,才把该作业的结果从打印机输出。

可见,在作业执行过程中不需要使用输入机和打印机。因此,系统只配置一台输入机和打印机的情况下,可让多个作业同时执行。

3. 实现技术

1) 输入井和输出井

为了实现虚拟设备,必须在磁盘上划出称为"井"的专用存储空间,用以存放作业的初始信息和作业的执行结果。为了便于管理,把井又分成输入井和输出井。输入井中存放作业的初始信息,输出井中存放作业的执行结果。

2) SPOOL 系统

操作系统中实现虚拟设备的功能模块是在计算机控制下通过联机的外部设备同时操作(Simultaneous Peripheral Operation On Line,SPOOL)来实现其功能的,所以,也把它称为SPOOL 系统。

SPOOL 系统由 3 部分程序组成:

(1) 预输入程序。人们经常把一批作业组织在一起形成作业流,预输入程序的任务是把作业流中的每个作业的初始信息传送到输入井保存以备作业执行时使用。

(2) 井管理程序。作业执行过程中要求启动输入机(或打印机)读文件信息(或输出结果)时,操作系统根据作业的请求调出井管理程序工作,转换成从输入井读信息(或把结果写入输出井)。对系统来说,从井中存取信息可以缩短信息的传输时间,从而加快作业的执行。

对用户来说,只要保证信息的正确存取就行,至于信息是从井中存取还是从独占设备上存取无关紧要。由于磁盘是可共享的,因此从井中存取信息可以同时满足多个用户的读/写要求,从而使每个用户都感到有供自己独立使用的输入机(或打印机)且速度与磁盘一样快。

井管理程序又可以分成井管理读程序和井管理写程序。当作业请求从输入机上读文件信息时,就把任务交给井管理读程序,从输入井读出信息供用户使用。当作业请求从打印机上输出结果时,就把任务转交给井管理写程序,把产生的结果保存到输出井中。

(3) 缓输出程序。缓输出程序负责查看输出井中是否有待输出的结果信息,若有,则启动打印机把作业的结果打印输出。

SPOOLing 系统的结构如图 7.15 所示。

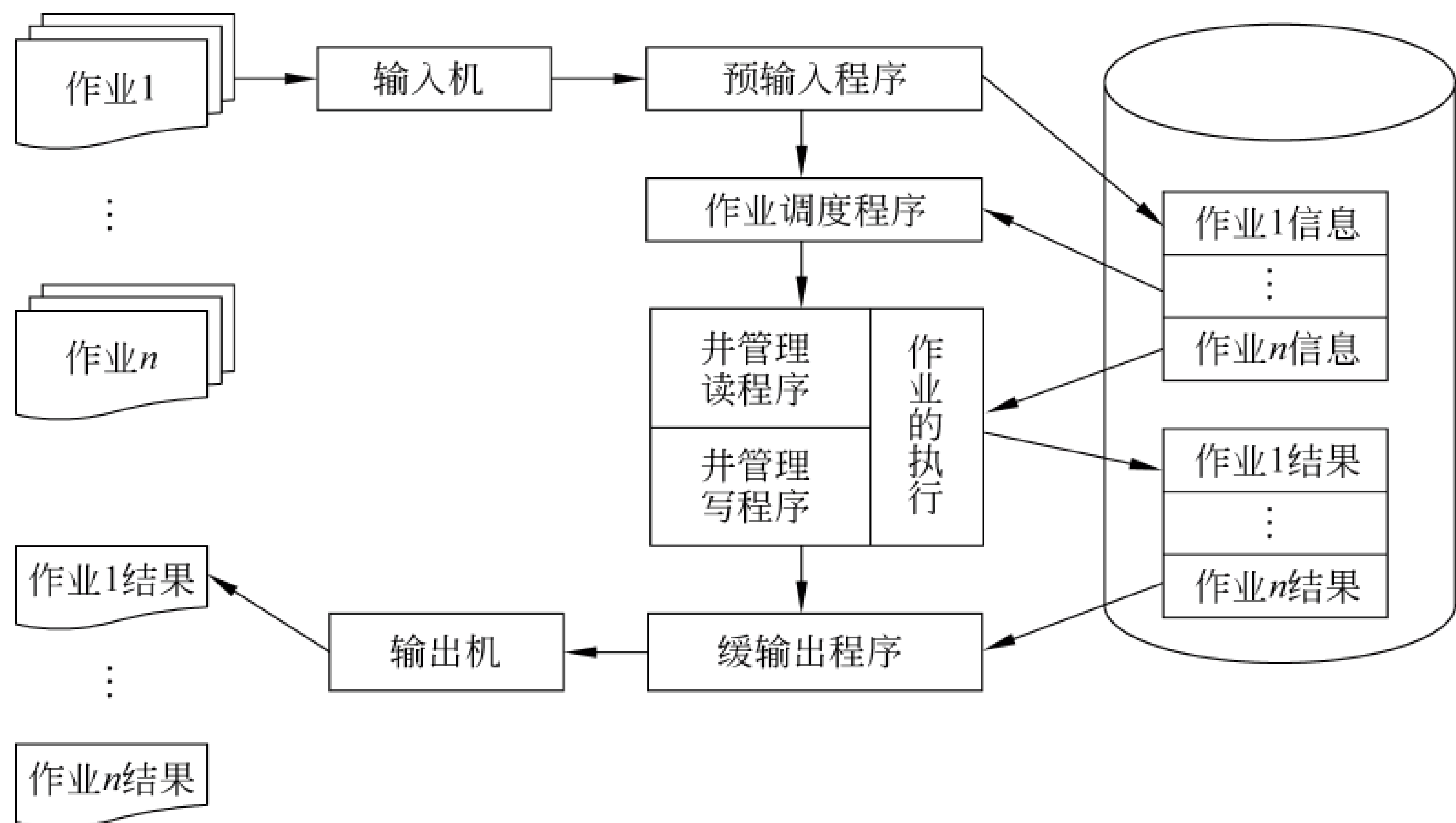


图 7.15 SPOOLing 系统的结构

3) 数据结构

为了能正确地管理进入系统的作业以及存取输入井和输出井中的信息,SPOOL 系统设计了 3 种数据结构:作业表、预输入表和缓输出表。

(1) 作业表。SPOOL 系统设置一张作业表,用来登记进入输入井的各个作业的作业名、作业状态、作业拥有的文件数以及预输入表和缓输出表的位置等。格式如表 7.4 所示。

表 7.4 作业表的一般形式

作 业 名	作业状态	文 件 数	其 他	预输入表位置	缓输出表位置

输入井中的作业可有 4 种状态:

① 提交状态。预输入程序启动了输入机,正在把该作业的信息传输到输入井。

② 收容状态。该作业的信息已经存放在输入井中,但尚未被选中执行。

③ 执行状态。作业已被选中并装入主存开始执行。

④ 完成状态。作业已执行结束,其执行结果在输出井中等待打印输出。

(2) 预输入表。每个作业有一张预输入表,用来登记该作业初始信息的各个文件。指出各文件的文件名、传输文件信息时使用的设备类型、文件的长度以及文件的存放位置等。

格式如表7.5所示。

表7.5　预输入表的一般形式

文　件　名	设　备　类	长　　度	其　　他	文件存放始址

作业的初始信息通常是源程序和数据等，作业执行时总是采用顺序存取方式。因此，把这些文件信息存入输入井时可采用链接结构，在预输入表中只需给出文件在输入井中的起始位置。

(3) 缓输出表。对每个作业设置一张缓输出表，用来登记该作业产生的结果文件。作业产生的结果也按链接结构组织成文件存放在输出井中，格式如表7.6所示。

表7.6　缓输出表的一般形式

文　件　名	设　备　类	长　　度	其　　他	文件存放始址

4) 功能实现

当用户提交了一批作业后，操作员输入预输入命令启动预输入程序工作。预输入程序查看作业表中是否有空登记项，若有空登记项则再检查输入井中是否有满足需要的空闲空间，如果有空闲空间则可接纳新的作业进入输入井。然后，启动输入机读出并分析作业的标识信息，把作业名和文件个数登记入作业表，且置作业为提交状态。接着依次读出作业的文件信息，寻找输入井中的空闲空间存放这些信息，把它们组织成链接文件的形式登记到预输入表中，直到该作业信息全部输入，把作业状态修改成收容状态，将预输入表的位置填入作业表。当作业流中还有后继作业时，预输入程序继续工作，只要能接纳新作业，就按上述过程把作业信息存入输入井，直到输入井的空间已占满或作业表中无空登记项时就暂不能再接纳新作业。当不能接纳新作业或当前作业流中信息已全部进入输入井，则预输入程序工作结束，当需要时可再次启动预输入程序工作。

当主存中可以装入作业时，就从输入井中选择处于收容状态的作业执行，被选中的作业其状态应改为执行状态。作业执行过程中要求启动输入机读文件时，系统并不实际地启动输入机，而是调出井管理读程序工作，根据作业名先找到该作业的预输入表，再根据文件名可从预输入表中得到文件存放的起始位置，沿着链接指针可依次读出存放在输入井中的文件信息。在这里，井管理读程序模拟从输入机读文件的工作。由于输入机把读出的信息传送给作业后就不再保留已读出的信息，所以，井管理读程序从输入井读出文件信息后也不必保留该信息。于是，当文件信息传送给作业后，应把文件占用输入井的存储空间归还，归还后的空间可以用来存放其他的文件。作业执行中要求启动打印机输出结果时，系统也不实际地启动打印机，而是调出井管理写程序工作。首先根据作业名找到缓输出表，同时查找输出井中的空闲空间，把结果信息组织成链接文件存入输出井，并在缓输出表中登记。作业执行结束后，把作业状态修改成完成状态。

当处理器空闲时，缓输出程序可以占用处理器，缓输出程序查看作业表，找出处于完成状态的作业，再查看相应的缓输出表得到结果文件的存放位置，读出文件信息并把它们转换

成符合打印要求的格式，然后启动打印机将其打印输出。文件信息被打印输出后，文件占用输出井的存储空间应归还。一个作业的结果文件均被输出后，应将其在作业表中除名，相应的登记项成为空登记项，可以用来登记新进入输入井的作业。

借助于硬件的中断装置和通道技术使得CPU与各种外部设备以及各外部设备之间均可并行工作。操作系统采用多道程序设计技术，合理分配处理器，实现联机的外部设备同时操作。作业执行时从磁盘上读/写信息来代替从输入机和打印机的读/写操作，不仅使多个作业可以同时执行，而且加快了作业的执行速度，提高了单位时间内处理作业的能力。在作业执行的同时还可利用输入机继续预输入作业信息和利用打印机输出结果。于是，整个系统可以是第一批作业的执行结果在打印输出，第二批作业正在处理，第三批作业信息正在预输入到磁盘的输入井中。这种联机同时操作极大地提高了独占设备的利用率，也使计算机系统的各种资源被充分利用。

7.8 I/O进程控制

7.8.1 I/O控制

引入I/O控制主要是为了解决系统在何时分配设备、在何时申请缓冲、由哪个进程进行中断响应、谁启动设备和谁设置I/O控制器中有关寄存器的值等问题。

通常把从用户进程的I/O请求开始，该用户进程分配设备和启动有关设备进行I/O操作，以及在I/O操作完成之后响应中断，进行善后处理为止的整个系统控制过程称为I/O控制过程。

7.8.2 I/O控制的功能

I/O控制的功能如图7.16所示。

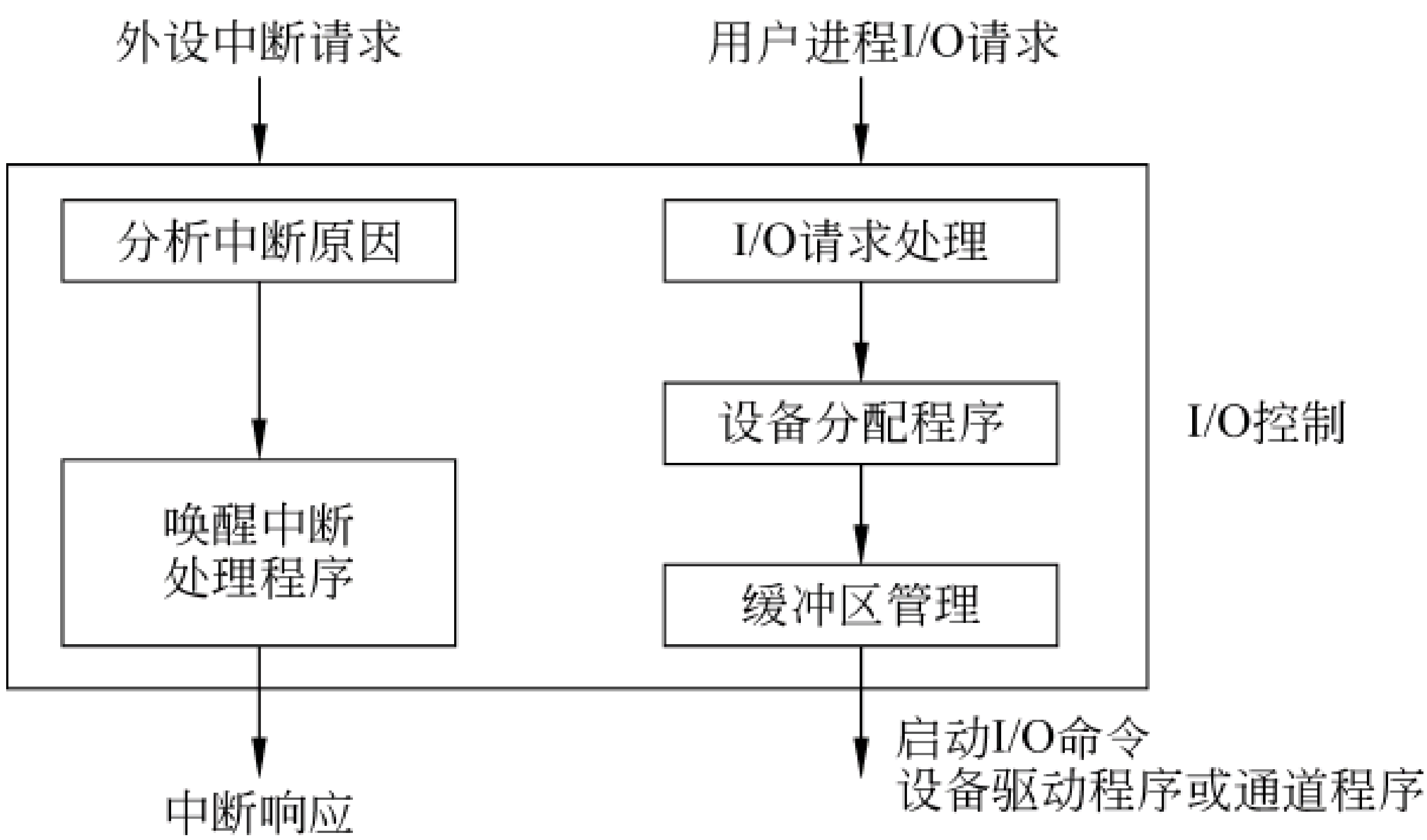


图7.16 I/O控制的功能

分析中断原因模块收集和分析调用I/O控制过程的原因是外设中断请求还是用户进程I/O请求，根据不同的请求，分别调用不同的程序模块进行处理。

I/O请求处理把用户进程的I/O请求变换为设备管理程序所能接受的信息。它首先将I/O请求中的逻辑设备名转换为对应的物理设备名，检查I/O请求命令中是否有参数错误，

如果没有错误,把该命令插入指向响应 DCT 的 I/O 请求队列,启动设备分配程序。在有通道的系统中,I/O 请求处理模块还将按请求命令的要求编制出通道程序。

在设备分配程序为 I/O 请求分配了相应的设备、控制器和通道之后,I/O 控制模块还将启动缓冲管理模块为此次 I/O 传送申请必要的缓冲区,以保证 I/O 传送的顺利进行。缓冲区的申请也可在设备分配之前进行。

在数据传送结束后,外设发出中断请求,I/O 控制过程将调用中断处理程序和做出中断响应。对不同的中断做不同的善后处理,例如释放设备、控制器和通道等。另外,还要检查是否有等待设备的 I/O 请求命令,如有,则要通知 I/O 控制进程进行下一 I/O 处理。

7.8.3 I/O 控制的实现

I/O 控制过程在系统中有 3 种实现方式:

(1) 作为请求 I/O 操作的进程的一部分实现。这种情况下请求 I/O 操作的进程应具有良好的实时性,且系统应能根据中断信号的内容准确地调度到请求所对应 I/O 操作的进程占据处理机,因为大多数情况下,当一个进程发出 I/O 请求命令之后,都被阻塞睡眠。

(2) 作为当前进程的一部分实现。在这种情况下不要求系统具有高的实时性。但由于当前进程与完成 I/O 操作无关,所以当前进程不能接受 I/O 请求命令的启动 I/O 操作。不过,当前进程可以在接收到中断信号后,将中断信号转交给 I/O 控制模块处理,因此,如果让请求 I/O 操作的进程调用操作控制部分,而让当前进程负责调用中断处理部分也是一种可行的 I/O 控制方案。

(3) I/O 控制由专门的系统进程——I/O 进程完成。在用户进程发出请求命令之后,系统调度 I/O 进程执行,控制操作。同样在外设发出中断请求之后,I/O 进程也被调度执行以响应中断。

I/O 进程包括请求模块、设备分配模块、缓冲区管理模块、中断原因分析模块、中断处理模块以及设备驱动程序模块。其实现方式也可分为 3 种:

(1) 每一类设备设置一个 I/O 进程且只能在系统态下执行,它专门执行这类设备的 I/O 操作。比如,为所有的交互式终端设置一个交互式终端进程,又如,为同一类型的打印机设置一个打印进程。

(2) 在整个系统中设置一个 I/O 进程,专门负责对系统中所有各类设备的 I/O 操作。也可以设置一个输入进程和一个输出进程,分别处理系统中所有各类设备的输入或输出操作。

(3) 每一类设备设置一个 I/O 进程,既能在系统态下执行,又能在用户态下执行,即为各类设备设置相应的设备处理程序(模块),供用户进程或系统进程调用。

7.9 设备驱动程序

设备驱动程序通常又称为设备处理程序,它是 I/O 进程与设备控制器之间的通信程序。又由于它常以进程的形式存在,故以后就统称为设备驱动进程。其主要任务是接收上层软件发来的抽象要求,如 read 或 write 命令,再把它转换为具体要求后,发送给设备控制器,启动设备去执行。此外,它也将由设备控制器发来的信号传送给上层软件。设备驱动程

序与硬件密切相关。

7.9.1 设备驱动程序的功能和特点

1. 设备驱动程序的功能

设备驱动程序的主要功能如下：

(1) 将接收到的抽象要求转换为具体要求。

(2) 检查用户I/O请求的合法性，了解I/O设备的状态，传递有关参数，设置设备的工作方式。

(3) 发出I/O命令，启动分配到的I/O设备，完成指定的I/O操作。

(4) 及时响应由控制器或通道发来的中断请求，并根据其中断类型调用相应的中断处理程序进行处理。

(5) 对于设置有通道的计算机系统，驱动程序还能根据用户的I/O请求自动地生成通道程序。

2. 设备驱动程序的特点

设备驱动程序与一般的应用程序及系统程序之间存在着下列的明显差异：

(1) 驱动程序主要是在请求I/O的进程与设备控制器之间的一个通信程序，它将进程的I/O请求传送给设备控制器，而把设备控制器中所记录的设备状态和I/O操作完成情况返回给请求I/O的进程。

(2) 驱动程序与I/O设备的特性紧密相关。因此，对于不同类型的设备，应配置不同的驱动程序。例如，可以为相同的多个终端设置一个终端驱动程序，但即使是同一类型的设备，由于生产厂家不同而并不完全兼容，因而也需分别为它们配置不同的驱动程序。

(3) 驱动程序与I/O控制方式紧密相关。常用的设备控制方式是DMA方式和中断方式，这两种方式的驱动程序明显不同，因为前者应按数组方式启动设备及进行中断处理。

(4) 由于驱动程序与硬件紧密相关。因而其中的一部分程序必须用汇编语言书写，目前有很多驱动程序，其基本部分已经固化在ROM中。

7.9.2 设备驱动程序的处理过程

不同类型的设备应有不同的设备驱动程序，但大体上可分成两部分，除了需要有能够驱动I/O设备工作的驱动程序外，还需要有设备中断处理程序来处理I/O完成后的工作。

设备驱动程序的主要任务是启动指定设备。但在启动之前还必须完成必要的准备工作。如检测设备是否"忙"等，在完成所有的准备工作后，才向设备控制器发送一条启动命令。设备处理程序的处理过程如下。

1. 将抽象要求转换为具体要求

通常在每个设备控制器中都有若干个寄存器，它们分别用于暂存命令、数据和参数等。用户及上层软件对设备控制器的具体情况毫无了解，因而只能向它们发出抽象的要求（命令），但又无法传送给设备控制器，因此，就需要能将这些抽象要求转换为具体要求。例如，将抽象要求中的磁盘块号转换为磁盘的柱面号、磁道号及扇区号。这一转换工作只能由驱动程序来完成，因为在操作系统中只有驱动程序才同时了解抽象要求和设备控制器中的寄存器情况，也只有它才知道命令、数据和参数应分别存入哪个寄存器。

2. 检查I/O请求的合法性

任何输入设备都只能完成一组特定的功能,如该设备不支持这次I/O请求,则认为这次I/O请求非法。例如,用户试图请求从打印机输入数据,显然系统应予以拒绝。此外还有些设备,如磁盘和终端,它们虽然是既可读又可写,但若在打开它们时规定的是读,则用户的写请求必须被拒绝。

3. 读出和检查设备的状态

要启动某个设备进行I/O操作,其前提条件应是该设备正处于空闲状态。因此在启动设备之前,要从设备控制器的状态寄存器中读出设备的状态。例如,为了向某设备写入数据,此时应先检查该设备的状态是否处于接收就绪,只有它处于接收就绪状态时,才能启动其设备控制器,否则只能等待。

4. 传送必要的参数

有许多设备,特别是块设备,除必须向其控制器发出启动命令外,还需传送必要的参数。例如,在启动磁盘进行读/写之前,应先将本次要传送的字节数、数据应到达的主存始址送入控制器的相应寄存器中。

5. 方式的设置

有些设备可具有多种工作方式,典型情况是利用RS-232接口进行异步通信。在启动该接口之前,应先按通信规程设定下述参数:波特率、奇偶校验方式、停止位数目及数据字节长度等。

6. 启动I/O设备

在完成上述各项准备工作后,驱动程序可以向控制器中的命令寄存器传送相应的控制命令。对于字符设备,若发出的是写命令,驱动程序将把一个字符数据传送给控制器;若发出的是读命令,则驱动程序等待接收数据,并通过从控制器中的状态寄存器读入状态字的方法来确定数据是否到达。

驱动程序发出I/O命令后,基本的I/O操作是在设备控制器的控制下进行的。通常I/O操作所要完成的工作较多,需要一定的时间,如读/写一个盘块中的数据,此时驱动程序进程把自己阻塞起来,直至中断到来时才将其唤醒。

7.9.3 设备驱动程序的管理

为了对驱动程序进行管理,系统中设置有设备开关表(Device Switch Table,DST)。设备开关表中给出相应设备的各种操作子程序的入口地址,例如打开、关闭、读、写和启动设备子程序的入口地址。一般来说,设备开关表是二维结构,其中的行和列分别表示设备类型和驱动程序类型。设备开关表也是I/O进程的一个数据结构。I/O控制过程为进程分配设备和缓冲区之后,可以使用设备开关表调用所需的驱动程序进行I/O操作。

7.10 Linux的设备管理

Linux沿用了UNIX处理设备的做法,把设备作为文件来处理,用户使用设备如同使用文件一样。

7.10.1 设备文件的概念

传统的 UNIX 系统均把设备当成文件来处理，因而可以用 read()和 write()对设备进行操作。设备文件一般在/dev 目录下，如/der/fd0 表示软盘，/dev/hda 表示第一个硬盘，/dev/hda1 表示第一个硬盘的第一个分区。

Linux 下的设备大体分为 3 类：

(1) 块设备。一次 I/O 操作以固定大小的数据块为单位，且可随机存取。

(2) 字符设备。一次 I/O 操作存取数据量不固定，只能顺序存取。

(3) 网卡。网卡是特殊处理的，它没有对应的设备文件。

设备文件除文件名之外，还有 3 个主要的属性：

(1) 类型。是字符设备还是块设备。

(2) 主设备号。主设备号相同的设备被同一设备驱动程序处理。

(3) 从设备号。用来指明具体的设备。

例如/dev/hdal，/dev/hdb2 都是块设备文件，主设备号均为 3，从设备号则分别为 1 和 2。Linux 安装完成之后已经在/dev 目录下生成了绝大多数可能要用到的设备文件，但很多相应的设备尚不存在。

7.10.2 相关数据结构

1. 字符设备管理

字符设备管理的主要数据结构如下：

```
struct device_struct{
    const char * name;
    struct file_operations * fops;
};
static struct device_struct chrdevs[MAX CHRDEV];
```

全局数组 chrdevs[]记录了所有字符设备的名称 name 及其对应的设备操作函数接口 fops，数组的下标则对应于设备的主设备号。

2. 块设备管理

块设备比字符设备管理要复杂一些，主要数据结构有 blkdevs[]和 blk_dev[]。

```
static struct{
    const char * name;
    struct block_device_operations * bdops;           /* 特定于设备的操作集 */
}blkdevs[MAX_BLKDEV];

struct blk_dev_struct{
    request_queue_t request_queue;                    /* 请求队列 */
    queue_proc * queue;
    void * data;
};
struct blk_dev_struct blk_dev[MAX_BLKDEV];
```

数组的下标依然对应设备的主设备号,其中 blkdevs[]通过 register_blkdev()函数初始化,而 blk_dev[]通过 blk_dev_init()函数初始化。blkdevs[]记录设备文件名及相应的操作集合,blk_dev[]记录各个设备的请求队列。

3. buffer cache

块设备的操作以块为基本单位,一般情况下,块的大小不会超过页面的大小。每读入一个块时,先在内核中申请一个大小相等的称为 buffer 的 cache,将块读入其中,然后再写入应用程序的缓冲区。这样,下一次访问该块时就不必进行磁盘操作,从而提高了性能。对块的写操作同样要经过 buffer 的 cache,每个 buffer 由 buffer_head 结构描述,该结构定义如下:

```
struct buffer_head{
    struct buffer_head * b_next;          /* 用来链接 hash 值相同的 buffer_head* /
    unsigned long b_blocknr;              /* 块号 * /
    unsigned short b_size;                /* 块的大小 * /
    kdev_t b_dev;                         /* 主设备号 * /
    kdev_t b_rdev;                        /* 次设备号 * /
    struct buffer_head * b_this page;     /* 同属一个页面的 buffer 链表 * /
    struct buffer_head * b_reqnext;       /* 同一个操作请求的 buffer_head 链表 * /
    struct buffer_head ** b_pprev;        /* 用来链接 hash 值相同的 buffer_head* /
    char * b_data;                        /* buffer 所在的位置 * /
    struct page * b_ page;                /* buffer 所属的页面 * /
    wait_queue_head_t b_wait;             /* 进程等待队列 * /
    struct inode*     b_inode;            /* 该 buffer 所属的 inode 结构 * /
};
```

每个 buffer 由设备号和块号唯一确定,并用这两者作为 hash 关键字快速地找到 buffer cache 的位置。通常,块的大小为 1KB,而物理页帧的大小为 4KB,所以一个物理页帧可以容纳 4 个 buffer。

4. 设备请求队列

Linux 对一个块操作的做法是,通常对每个块设备建立一个请求队列,该队列的每一个成员为操作请求。操作请求结构的定义如下:

```
struct request{
    int cmd;                              /* 操作行为:读或写 * /
    struct buffer_head * bh;              /* buffer_head 链表头 * /
    struct buffer_head * bhtail;          /* buffer_head 链表尾 * /
};
```

每个操作请求都对应着一个 buffer_head 链表。每当需要对一个块进行操作时,要将相应的 buffer_head 加入设备请求队列。

7.10.3 中断和异常

中断通常分为同步中断和异步中断。同步中断由 CPU 产生,Linux 使用 int 指令来

实现系统调用(即同步中断)。异步中断又分为可屏蔽的和不可屏蔽的两类,由一些硬件设备产生,可以在指令执行的任意时刻产生,I/O设备和定时器产生的中断就属于异步中断。

Intel处理器手册把同步中断称为异常,把异步中断直接称为中断。

每个中断/异常都有一个向量号,该号的值为0～255,该值是中断/异常在中断向量表(Interrupt Descriptor Table,IDT)中的索引,每个中断/异常均有其相应的处理函数,中断/异常在使用前,必须在IDT中注册信息,以保证发生中断/异常时能找到相应的处理函数。IDT表项还记录了一些其他信息用于安全检查以防止非法进入核心态。IDT在系统初始化时创建。

IDT中向量号的使用情况如下:

(1) 异常与非屏蔽中断使用0～31。

(2) 可屏蔽中断使用32～47。

(3) 剩下的索引号中,Linux内核只使用了128(0x80)来实现系统调用。

7.10.4 Linux的设备驱动程序

1. 设备驱动程序的功能和特征

Linux内核的设备管理是由一组运行在特权级上、驻留在内存的对底层硬件进行控制和处理的共享库的驱动程序完成。

Linux的设备驱动程序的主要功能如下:

(1) 对设备进行初始化。

(2) 使设备投入运行和退出。

(3) 从设备接收数据并将其送入内核。

(4) 从内核将数据送到外设。

(5) 检测和处理设备完成任务和出错情况。

在Linux中,设备驱动程序是一组相关函数的集合,含设备服务子程序和中断处理程序。设备服务子程序包含了所有与设备相关的代码。设备驱动程序利用结构file_operations与文件系统联系,因为对设备的各种操作的入口参数都放在file_operations中。对于一些特定的设备,其入口为NULL。

Linux中,不同的设备驱动程序有一些共同的特征:

(1) 驱动程序都属于内核代码。

(2) 为内核提供了统一接口。

(3) 驱动程序的执行属于内核机制并使用内核服务。

(4) 驱动程序可以动态地以模块形式加载或卸载。

2. 设备驱动程序的框架

设备驱动程序是内核的一部分,但是由于设备种类繁多,相应地,设备驱动程序的代码也多,而且设备驱动程序往往有很多人来开发。为了能协调设备驱动程序和内核之间的开发,就必须有一个严格定义和管理的接口。例如,SVR4提出了DDI/DKI(Device-Driver Interface/Drive-Kernel Interface)规范。通过它可以规范化设备驱动程序与内核之间的

接口。

Linux 的设备驱动程序与外设的接口与 DDI/DKI 规范相似,可分为3部分:

(1) 驱动程序与系统引导的接口。这部分通过数据结构 file_operations 完成。

(2) 驱动程序与设备引导的接口。这部分利用驱动程序对设备进行初始化。

(3) 驱动程序与设备的接口。这部分描述了驱动程序如何与设备进行交互,这与具体设备相关。

根据功能要求,设备驱动程序的代码可分为如下几个部分:

(1) 驱动程序的注册与注销。

(2) 设备的打开与释放。

(3) 设备的读/写操作。

(4) 设备的控制操作。

(5) 设备的中断和轮询处理。

3. 设备驱动程序的几个通用函数

1) open()函数

其功能是确定有关硬件是否可供使用并且已经联机,验证从设备号是否有效。如果每次只允许一个进程打开这个设备,设置忙标志,成功返回0,否则返回负数。

2) release()函数

其功能是如果有未结束的I/O,则对它们完成清理工作。

3) ioctl()函数

除了基本的操作外,有时还需要传送控制信息给设备驱动程序,或从它那里取得状态信息,这时就用 ioctl()函数。

7.11 本章小结

本章首先介绍了设备的分类、设备管理的功能和任务以及设备同主机进行数据交换的方式。

磁盘是主要的外存储设备,本章简单介绍了磁盘的结构,磁盘驱动调度由移臂调度和旋转调度两部分组成,常用的移臂调度算法有先来先服务、最短寻找时间优先、单向扫描和电梯调度。信息在磁盘上优化分布可以缩短I/O操作时间。

中断技术是实现外设与主机并行工作的基础,采用通道技术能实现外设与CPU高度并行。缓冲技术缓解了外设速度与主机速度的差异问题。

设备分配方式有两种:静态分配和动态分配。静态分配方式不会出现死锁,但设备的使用效率低。动态分配方式有利于提高设备的利用率,但如果分配算法使用不当,则有可能造成进程死锁。常用的分配策略有先请求先分配和优先级高者先分配策略。设备分配中使用的数据结构有设备控制表、系统设备表、控制器控制表和通道控制表,它们之间具有一定的关联。

操作系统利用共享设备来模拟独占设备的工作,以提高独占设备的使用效率,这种技术为用户提供了虚拟设备,通常称为SPOOLing系统。

I/O控制和设备驱动程序也是设备管理中的任务。

本章最后介绍了Linux系统的设备管理方法。

习　　题

1. 从哪几个角度对设备进行分类？每种分类中把设备分为哪几类？

2. 简述设备管理的功能。

3. 数据传输的方式有哪几种？

4. 常用的磁盘调度算法有哪几种？

5. 假定某磁盘共有200个柱面，编号是0～199，如果在为访问143号柱面的访问者服务后，当前正在为125号柱面的请求服务，同时有若干访问者早已等待服务，它们访问的柱面号依次是86、147、91、177、94、150、102、175、130。

请回答下面的问题：

(1) 分别用先来先服务、最短寻找时间优先、电梯调度和单向扫描算法来确定实际的服务满足次序。

(2) 按实际服务次序分别计算上述算法下移动臂移动的距离和平均寻道长度。

(3) 假设在寻道时，移动一个柱面需要6ms，按实际服务次序分别计算上述算法下总的寻道时间。

6. 假定某磁盘的旋转速度是20ms/转，格式化时每个盘面被分成个10扇区，现有10个逻辑记录存放在同一磁盘上，安排如图7.17所示。处理程序要顺序处理这些记录，每读出一条记录后处理程序要花4ms的时间进行处理，然后再顺序读下一条记录并进行处理，直到处理完成这些记录，回答下面的问题：

(1) 顺序处理完这10条记录总共花费了多少时间？

(2) 请给出一种记录优化分布的方案，使处理程序能在短时间内处理完这10条记录，并计算优化分布时需要花费的时间。

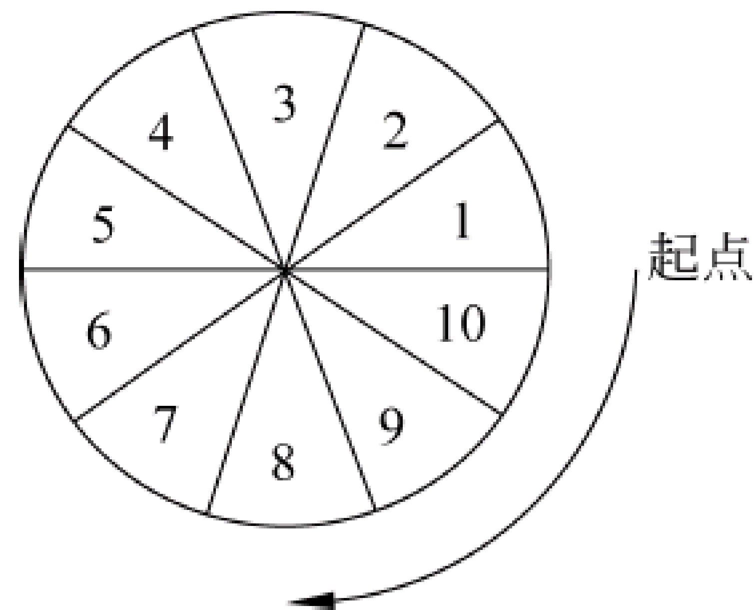


图7.17 逻辑记录存放次序

7. 若现在磁盘的移动臂处于第15号柱面有6个请求(如表7.7所示)等待访问磁盘，如何响应这些访问才最省时间？

表7.7 6个请求的访问要求

序号	柱面号	磁头号	扇区号	序号	柱面号	磁头号	扇区号
1	12	2	6	4	6	4	1
2	5	3	2	5	16	7	3
3	16	8	7	6	12	5	6

8. 假定磁盘的存取臂现在处于6号柱面上，有如表7.8所示的6个请求等待访问磁盘，试列出最省时间的响应顺序。

表 7.8　6 个请求的访问要求

序号	柱面号	磁道号	扇区号	序号	柱面号	磁道号	扇区号
1	7	6	3	4	7	4	4
2	5	5	6	5	20	9	3
3	15	20	6	6	5	15	2

9．假设一个磁盘组共 100 个柱面，每个柱面上有 8 个磁道，每个盘面被分成 8 个扇区。现有一个含有 6400 个逻辑记录的文件，逻辑记录的大小与扇区一致，该文件以顺序结构的形式被存储到磁盘上。柱面、磁道和扇区的编号从 0 开始，逻辑记录的编号也从 0 开始。文件信息从 0 柱面、0 磁道、0 扇区开始存放，试问：

（1）该文件的第 3680 个逻辑记录应该存放在什么位置？

（2）第 78 柱面的第 6 磁道的第 6 扇区中存放了该文件的第几个逻辑记录？

10．为什么引入中断？解释以下概念：中断、中断处理、中断响应、关中断、开中断、中断屏蔽。

11．为什么引入通道？什么是通道？通道有几种类型？各适用于哪些场合？I/O 系统的三连四空指的是什么？通道程序是如何生成和执行的？

12．为什么引入缓冲技术？缓冲有哪几种类型？缓冲池是如何构成的？以数据输出为例说明对缓冲池中的各个队列的操作。

13．简述设备分配的原则和策略。在设备分配中使用了哪几种数据结构？它们之间的关系如何？

14．设备的独立性指的是什么？独占设备是如何分配的？

15．为什么引入虚拟设备？实现设备的基本条件有哪些？

16．简述 SPOOL 系统的工作原理。

17．为什么是 I/O 控制？它的主要功能是什么？它有哪几种实现方式？

18．简述设备驱动程序的功能和特点。如何管理设备驱动程序？

19．Linux 中把设备作为文件有何特点？

20．简述 Linux 设备驱动程序的主要功能。设备驱动程序的代码由哪几部分组成？

21．简述 Linux 设备管理中涉及的主要数据结构有哪些。

22．简述 Linux 中的中断和异常。

第 8 章　进程的互斥、同步、通信和死锁

进程和线程的管理是现代操作系统的主要功能，进程的互斥、同步、通信和死锁是进程管理所要解决的首要问题。现代计算机系统无论是仅有一个 CPU 的单机系统，还是具有多个 CPU 的多处理器系统或分布式系统，均支持多道程序设计。这样系统内就存在多道程序，即多个进程在同时运行，这就是所谓的并发性（concurrency）。这种并发性具体涉及如下几方面问题：进程间的协作和通信、对资源的共享和竞争、多个进程活动间的同步、进程间 CPU 时间的分配等。在这种并发环境下，即使是在单机系统内也可能存在多个进程同时存在，因此就可能出现多个进程同时争夺同一个共享资源的可能性。系统中一般存在两种共享资源，一种共享资源允许多个进程同时访问，如磁盘文件；而另一种共享资源不允许多个进程同时访问，即进程对这些资源的访问是互斥的，如打印机。对于仅允许互斥访问的资源，如果不加以控制地进行访问则必然会引起系统出错，甚至可能导致进程互相占用对方想要的资源而使系统不能正常地运行，即死锁现象的发生。同时可以看出，存在于系统内的多个进程间可能会发生某种联系，它们可能相互协作共同完成同一项任务；这些协作进程的执行可能存在时序上的关系，因而这些进程的执行需要进行同步；而协作进程间可能需要交换必要的信息，因而进程间要有一定的通信机制。

下面从资源共享开始，介绍进程互斥和同步的基本概念和实现方法、死锁的基本概念和预防措施以及进程间的通信机制。

8.1　进 程 互 斥

众所周知，计算机系统中的资源是有限的，为了提高系统资源的利用率，同时满足多个并发进程运行的要求，操作系统中引入了资源共享机制，允许多个进程同时访问系统中的共享资源。系统中有些资源是不允许同时访问的（如打印机），并发进程对这些资源的使用是轮流进行的。也就是当某个进程使用这些资源时，其他进程则不能使用，即各并发进程对这些资源的使用是互斥的。人们把这种某一时刻仅允许一个进程访问的共享资源称为临界资源（critical resource），使用临界资源应当是互斥的。

8.1.1　临界区与进程互斥

具有并行特征的诸多进程在同一环境中运行，由于系统资源有限，不可避免地导致对共享资源的激烈竞争。当若干个进程均要使用同一临界资源时，它们必须以互斥的方式进行。人们把程序中使用某一临界资源的那段代码称为程序的临界区（critical section）。可见临界区是相对于某一临界资源的，而且在某一时刻仅允许一个程序处于临界区。

一般来说，可以把临界区按不同的临界资源划分为不同的集合，把这些集合称为类（class）。同一类中的临界区称为相关临界区。相关临界区由于使用同一临界资源，一般来说是不允许交叉执行的，由共享公有资源而造成的对并发进程执行速度的制约称为间接制约。

下面通过一个例子形象、直观地说明临界资源、临界区和相关临界区等概念。

在使用C语言等高级语言进行程序设计的过程中，常常需要动态地申请和释放内存，以提高内存的利用率。现假设空闲内存块的首地址是以堆栈的形式进行组织管理的，栈顶指针Top指出第一个空闲内存块的首址，如图8.1所示。下面定义两个过程askspace()和releasespace()，分别完成空闲内存块的申请和释放，这两个过程具体实现如下：

```
/* 申请由栈顶指针top所指内容为首址的一个空闲内
   存块 */
askspace()
{
    char * add;
    add←stack(top);
    top←top-1;
    return(add)
}
/* 把释放的空闲内存块首址压入栈顶 */
releasespace(address)
{
    top←top+1;
    stack(top)←address;
}
```

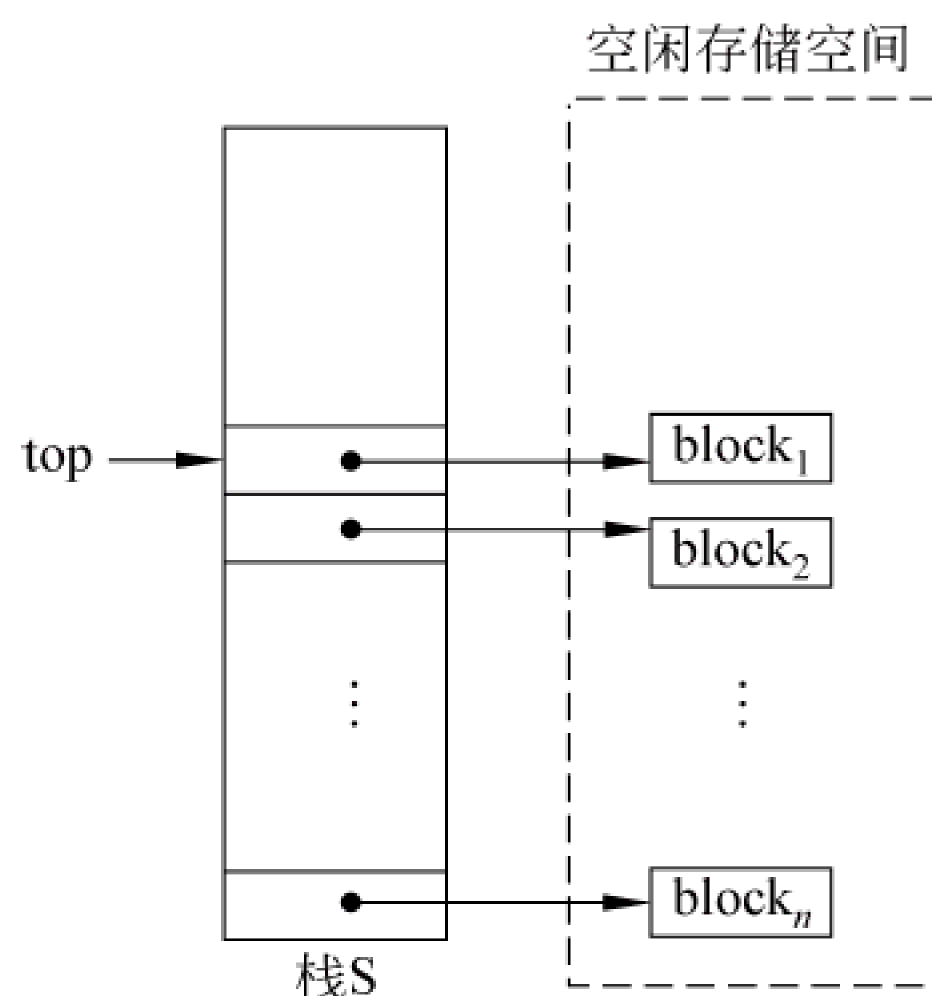


图8.1 空闲内存块的组织

当过程askspace()申请空闲存储块时，首先申请得到由栈顶指针top所指单元内容为地址的存储块。当过程releasespace(address)释放一个首地址为address的空闲存储块时，则需将该存储块首地址压入栈S。设与这两个过程相对应的进程分别为Pa和Pr，显然，堆栈S为这两个进程共同操作的对象。

当这两个进程相互交叉执行时，各进程是否能够正确完成各自的功能呢？设时刻t0时，top＝s0，则askspace()和releasespace(address)可能按如下顺序执行：

首先执行releasespace(address)的第一句：

```
t0: top←top+1;                /* top=s0+1 */
```

接着askspace()再执行：

```
t1: add←stack(top);           /* add=stack(s0+1) */
t2: top←top-1;                /* top=s0 */
```

然后执行releasespace(address)的第二句：

```
t3: stack(top)←address;       /* stack(s0)←address */
```

可见，执行结果是调用askspace()的进程取到的是s0＋1中未定义的值，而调用releasespace(address)的进程把释放的空存储块address放入了s0中，覆盖了原栈顶单元中的有用地址信息，致使丢掉了一个空闲存储块。

显然，与askspace()和releasespace(address)相对应的两个进程是不可以交叉执行的，即这两个进程不允许并发执行，每个进程应是一个完整的动作序列。此处的堆栈S就是临界资源，而askspace()和releasespace(address)则是相对于临界资源(堆栈S)的临界区，这

两个程序段共享了堆栈 S 中的公共数据,即空闲存储块的首址,它们是相关临界区,这两个临界区组成一个类 sp,即 sp={askspace,releasespace};这两个进程由于共享了堆栈 S 中的公共数据,而相互之间存在着制约,即一个的执行制约了另一个的执行,它们之间这种由于共享同一公共数据而导致的制约即为间接制约。

因此,临界区也可以定义为不允许多个并发进程交叉执行的程序段,也称为临界部分。显然,临界区是由于属于不同并发进程的程序段共享公用数据而引起的,有时临界区也称为访问公共数据的程序段。

由上面的分析可知,因共享资源而导致了进程执行过程的相互制约。一般进程间的相互制约关系有两种:一种是进程间的间接制约,它是由于并发进程共享某一公有资源而引起的相互制约关系。例如,两个进程共享同一台打印机,一个进程使用过程中另一进程就不能使用,否则两个进程的输出内容交织在一起输出,结果混乱将无法辨认;只能是一个进程执行完成,打印出完整结果后,另一进程才能使用打印机。显然,这两个进程因共享打印机而产生了相互制约,这种关系即为间接制约。另一种制约关系称为直接制约,它是因为并发进程一方的执行结果互为对方的执行条件,从而限制了各进程的执行速度。进程间的间接制约对应进程间的互斥,进程间的直接制约对应进程同步。

显然,临界区的进入是非常严格的,否则将得不到正确的执行结果。对于每一类临界区,系统应有一套完整的措施和方法分配和释放公共资源,以制约并发进程的执行,这就是互斥。

由以上分析可见,进程互斥也可以理解为不允许两个以上的共享公共资源的并发进程同时进入临界区。

为了使系统中的并发进程能够正确地访问它们所共享的临界资源,进程互斥执行时必须满足如下准则:

(1) 在共享同一临界资源的多个并发进程同时申请进入临界区时,每次仅允许一个进程进入。

(2) 当多个并发进程同时申请进入临界区时,应在有限时间内让一个进程进入临界区,而不应相互堵塞,以至于各个进程都不能进入临界区。

(3) 不应使要进入临界区的进程无期限地等待在临界区之外。而进程一旦进入临界区,则只能在临界区内逗留有限时间,以释放临界资源,满足其他并发进程的分配请求。

(4) 在临界区外运行的进程不应阻止其他并发进程进入临界区。

(5) 当没有进程处于临界区时,任何要求进入临界区的进程应允许立即进入。

(6) 不能假设各并发进程的相对执行速度以及可用处理器的数目。各并发进程是平等的、独立的,若某一进程从临界区退出,其他并发进程均可进入临界区。

总之,各并发进程享有平等、独立地竞争和使用临界资源的权利,但要保证某一时刻只能有一个进程处于临界区内,即这些并发进程应该互斥地执行。

8.1.2 互斥的加锁实现

有关临界区的互斥访问技术方面的研究开始于 20 世纪 60 年代。Dijkstra、T. Dekker 和 Knuth 等科学家先后提出了许多算法用于解决临界资源的互斥访问。最简单的途径是采用对临界区加锁的方法来实现并发进程的互斥。

这种临界区加锁的方法具体为：当某个进程进入临界区后，它将锁上临界区，直到它退出临界区时为止，此时其他进程是不能进入临界区的。当某个并发进程想要进入临界区时，首先测试该临界区是否能加上锁，若临界区已被锁住，则该进程反复测试直到该临界区解锁后才可能进入。

设临界区的类名为 S，key[S]表示定位于类名为 S 的临界区上的一把锁。加锁后的临界区表示如下：

```
lock(key[S]);
<临界区>
unlock(key[S]);
```

上面用到了两个原语：lock(key[S])和 unlock(key[S])，它们分别用于对临界区进行加锁和解锁操作。key[S]用于表示施加于临界区 S 上的一把锁，当 key[S]=1，表示锁已打开，进程可以进入临界区；当 key[S]=0，表示锁已加上，并发进程暂时不能进入临界区，应等待相应临界资源的释放。

加锁原语 lock(key[S])和解锁原语 unlock(key[S])的具体实现如下：

```
lock(key[S])
{
    int v;

    repeat
    v←key[S];
    step1: until v=1
    step2: key[S]←0;
}
unlock(key[S])
{
    key[S]←1;
}
```

为了防止在 step1 判断 v=1 为真后在执行 step2 之前其他并发进程进入临界区，有些计算机在硬件中设置了“测试与设置”(test and set)指令，使得 step1 和 step2 不可分离。

通过加锁原语 lock(key[S])和解锁原语 unlock(key[S])的实现可以看出，加锁原语中反复测试锁状态浪费了许多 CPU 时间，在并发进程数目多时更为突出。另外，并发进程进入临界区时自己调用加锁原语 lock(key[S])判断是否可以进入临界区，一旦进入就可能长时间循环占用临界区而阻碍其他并发进程进入临界区，从而导致不公平现象的出现。

8.1.3 信号量和 P、V 原语

为了克服加锁实现互斥中存在的问题，荷兰学者 Dijkstra 提出了信号量和 P、V 原语的概念。信号的概念出现于交通领域，它通过信号颜色的变化进行交通管理。而操作系统中的两个或多个进程则借助信号进行协同运行，这样就可以强制一个进程停止在一个规定的位置上，直到它接收到一个特定的信号才能被唤醒。为了表示这种信号，操作系统中采用了

称为信号量的变量。信号量是一个整数,在一般情况下,当信号量大于零时表示可供使用的共享资源数,当信号量小于零时表示正在等待使用临界资源的进程数。下面通过一个例子对信号量的含义进行形象化的说明。

设有 m 个学生准备到公用教室学习,该公用教室共有 n 个座位,且 n 不大于 m。令 sem 表示该教室内的空座位数,其初值为 n。显然对于学生来讲,公用教室中的座位是他们共享的临界资源,每个座位仅允许被一个学生占用,而学生则相当于并发进程,sem 相当于信号量。当有一名学生想要进入教室时,首先 sem 减 1,若减 1 后 sem 不小于零时,说明有空座位,此时学生则可直接进入教室;若减 1 后 sem 小于零,则说明无空座位,学生不能进入教室,此时 sem 表示等待进入教室的学生数。当有一个学生从教室中离开时,sem 应加 1,表示可用的座位增加一个,此时若 sem 不小于零,说明有学生正在等待进入教室,则允许一个学生进入。

由此可见,信号量定义为一个整形变量,大于零时表示可供并发进程使用的共享资源数,小于零时表示正在等待使用临界资源的并发进程数。信号量只能由 P、V 原语对其进行操作,由此实现临界区的互斥进入。

并发进程进入临界区的具体过程为:当并发进程进入临界区前需调用 P 原语判断是否可以进入临界区,只有允许才能进入;而并发进程在退出临界区时需调用 V 原语声明已释放一个临界资源,以通知其他并发进程可以进入临界区。

采用 P(S)、V(S)原语,进入类名为 S 的临界区的过程描述如下:

```
{
    When S do {
    P(sem);
    <临界区>
    V(sem);
    }
}
```

其中 sem 为信号量。

P 原语的具体实现过程如下:

(1) sem=sem-1。

(2) 若 sem≥0,则进程继续进行。

(3) 若 sem<0,则该进程调用阻塞原语阻塞自己,然后进入与该信号相对应的等待队列中,并转进程调度程序。

可见 P 原语检测到不能进入临界区时不像 lock()操作反复进行测试,而是调用阻塞原语阻塞自己,并进入等待队列,等待其他进程执行 V 原语操作释放资源后,再申请进入临界区。

V 原语的具体实现过程如下:

(1) sem=sem+1。

(2) 若 sem>0,则进程继续进行。

(3) 若 sem≤0,则从该信号所对应的等待队列中唤醒一个等待进程,然后返回原进程继续执行或转向进程调度程序。

8.1.4 利用P、V原语实现进程互斥

利用P、V原语和信号量可以方便地实现并发进程的互斥操作,并可避免出现加锁法所引起的问题。下面通过事例进一步说明P、V原语和信号量的使用。

例8.1 设信号量sem的初值为1,只要把临界区置于P(sem)和V(sem)之间,即可实现n个并发进程p(i)间的互斥执行。具体实现算法如下:

```
/* mutualexclusion program* /
const n= the total number of processes;
semaphone sem;
sem= 1;                          /* sem为信号量,此时临界区内仅存在一个共享资源 * /
procedure p(i:integer)           /* i∈ {1,2,…,n} * /
{
    P(sem);
    <临界区 >
    V(sem);
}
/* main program* /
main()
{
    cobeging
        p(1);
        p(2);
        ⋮
        p(n);
    coend
}
```

例8.2 设仅有一个属于类名为S的临界资源,供两个并发进程Pa和Pb使用,请利用信号量实现这两个并发进程的互斥执行。

解:(1) 设sem为互斥信号量,其取值范围为(1,0,−1)。其中信号量sem各种取值的意义如下:

sem=1表示两进程均未进入类名为S的临界区。

sem=0表示仅有其中一个进程处于类名为S的临界区。

sem=−1表示其中一个进程已进入类名为S的临界区,另一个进程正等待进入该临界区。

(2) 进程Pa()和Pb()的描述如下:

```
Pa()                             /* 进程 Pa * /
{
    P(sem);
    <临界区>
    V(sem);
    ⋮
```

```
}
Pb()                              /* 进程 Pb */
{
    P(sem);
    <临界区>
    V(sem);
     ⋮
}
```

同样,当存在 n 个并发进程时,信号量 sem 的取值范围及其意义如下:

sem=1 表示无进程在临界区。

sem=0 表示有一个进程处于临界区。

sem=-1 表示有一个进程处于临界区,而且有一个进程正等待进入临界区。

sem=-(n-1)表示有一个进程处于临界区,而且有 n-1 个进程正等待进入临界区。

此时,信号量被初始化为 1,表示仅存在一个共享资源;当存在 m 个共享资源时,信号量 sem 可初始化为 m,但此时信号量 sem 值的意义已发生变化。

8.2 进程同步

8.2.1 进程同步的概念

在计算机系统中执行的协作进程彼此是相互影响的,一方面它们彼此协作以完成共同的任务,另一方面它们又相互竞争有限的系统资源。这种相互协作和资源竞争就导致并发进程间的相互制约。例如,系统内存在两个进程:计算进程和打印进程,它们公用一个打印缓冲区,如图 8.2 所示。计算进程将其计算结果送入打印缓冲区中,打印进程从中取出并送到打印机上打印出来。显然计算进程和打印进程为一对协作进程,它们共同协作完成待求解问题的计算和打印工作。打印缓冲区为两个协作进程的共享资源。可以看出,打印进程运行的先决条件是计算进程完成了计算工作并将计算结果送到了打印缓冲区。如果打印缓冲区内无结果数据,打印进程则无数据可打印,因而就不能执行。反之,当打印缓冲区中的数据没有被打印进程取空之前,计算进程就不能向缓冲区写入新的数据。那么计算进程和打印进程如何协调工作才能顺利完成共同的任务呢?实现这两个进程协调工作的一种途径为:当缓冲区为空时,计算进程可以执行并将计算结果送入打印缓冲区,然后它向打印进程发送信号通知打印进程可以打印数据;打印进程取走打印缓冲区中的数据后,它向计算进程发送信号通知计算进程打印缓冲区已空,计算进程可以继续执行。显然,计算进程和打印进程的执行是互为条件的,这时两个进程间的关系为直接制约关系。



图 8.2 计算进程和打印进程协同工作示意图

如 8.1 节所述,在并发环境下,实现进程互斥操作的原则之一是不能假设各并发进程的相对执行速度,即各并发进程开始执行的时间是随机的,其执行速度是独立的,此时并发进程所处的环境称为异步环境。

这样,异步环境下的一组并发进程因直接制约而互相合作,使得各进程按一定的顺序和速度执行的过程称为进程间的同步。具有同步关系的一组并发进程称为合作进程。合作进程间相互发送的信号称为消息或事件。这样,同步的各进程之间可以通过发送消息来通知其他进程执行所需的条件是否具备,从而达到各协作进程同步运行的目的。

8.2.2 进程同步的实现——消息发送

消息发送法是根据进程同步的概念提出的一种实现进程间同步的一种方法。用该方法实现进程同步需定义两个过程:

(1) wait(消息名):进程等待合作进程发来消息。

(2) signal(消息名):进程向合作进程发送消息。

下面以计算和打印问题来说明这种方法的具体实现。

设 Pc()为计算进程,用于完成计算工作,并将计算结果存入缓冲区。执行保存结果的操作之前需等待缓冲区空的消息。若等到了缓冲区空的消息,则继续执行;否则继续等待该消息。

设 Pp()为打印进程,它从缓冲区中取数据,并将数据送到打印设备上打印;执行前需等待缓冲区中有数据的消息。若等到了缓冲区中有数据的消息,则继续执行;否则继续等待该消息。

为此设消息名 Bufempty 表示缓冲区空,消息名 Buffull 表示缓冲区中有数据。Bufempty 初值为 true,Buffull 初值为 false。

进程 Pc()和 Pp()的具体描述如下:

```
Pc()                              /* 计算进程 Pc */
{
    while(true){
        result←Calculate();
        wait(Bufempty);
        Buf←result;
        Bufempty←false;
        signal(Buffull);
    }
}
Pp()                              /* 打印进程 Pp */
{
    while(true){
        wait(Buffull);
        result←Buf 的内容;
        clear(Buf);
        Buffull←false;
        signal(Bufempty);
        print(result);
    }
}
```

注意此时,wait()等待消息名为 true 后继续执行,signal()向合作进程发消息并置消息名为 true。由上面的讲述可以看出,各进程之间发送的消息也可看做信号量,因此进程的同

步也可以使用 P、V 原语和信号量来实现。

8.2.3 进程同步的实现——P、V 原语和信号量

下面仍以计算和打印问题来说明这种方法的具体实现。

设 Pc()为计算进程,用于完成计算工作,并将计算结果存入缓冲区。执行保存结果的操作之前需判断缓冲区空否,若缓冲区满,则等待,否则执行。

设 Pp()为打印进程,它从缓冲区中取数据,并将数据送到打印设备上打印。执行前需判断缓冲区是否为空,若缓冲区空,则等待,否则执行。

设两个信号量:Bufempty 和 Buffull。其中 Bufempty 表示缓冲区为空否,初值为 1,表示缓冲区为空;Buffull 表示缓冲区中是否有数据,初值为 0,表示缓冲区中无数据。

进程 Pc()和 Pp()的具体描述如下:

```
Pc()                              /* 计算进程 Pc */
{    while(true){
         result←Calculate();
         P(Bufempty);
         Buf←result;
         V(Buffull);
     }
}
Pp()                              /* 打印进程 Pp* /
{
     while(true){
         P(Buffull);
         result←Buf 的内容;
         clear(Buf);
         V(Bufempty);
         print(Buf);
     }
}
```

由上面的讲述可以看出,用于同步的信号量仅与需同步的进程有关,与其他进程无关。通常把进程同步使用的信号量称为该进程的私有信号量,把互斥时使用的信号量称为公有信号量。当一个过程包括若干个公有、私有信号量时,P、V 原语操作的顺序相当重要。一般 V 原语操作释放资源,出现次序可以任意;但 P 原语则不然,若使用次序混乱,则会造成系统死锁(deadlock)。死锁的概念将在 8.5 节中讲述。

8.2.4 进程同步的实现——管程

尽管信号量为进程同步提供了一种既方便又简单有效的实现机制,但每个要访问临界资源的进程自身都必须显式地给出 P、V 操作。这样就使大量的同步操作分散在多个并发进程中,这不仅给编程造成了麻烦,也给系统的管理增加了负担。而且信号量的不正确使用可能导致很难检测的定时错误,尽管这些错误是由于执行一些不常发生的特定序列所引起的。为了克服上述问题,1971 年 Dijkstra 提出一种方法,即把所有对某一临界资源的互斥

和同步操作集中起来,构成一个“秘书”进程,对于该临界资源的所有访问都要通过该进程来实现。

1975 年 Hansen 和 Hoare 将“秘书”进程的思想发展成管程(monitor)的概念,为实现进程同步提供了一种新机制。Hansen 所做的管程定义为“一个管程定义了一个数据结构和能为并发进程调用的在该数据结构上进行操作的一组过程,这组过程能同步进程和改变管程中的数据。”显然,一个管程包括 3 部分:局部用于管程的公共变量说明,对该数据结构进行操作的一组互斥执行的过程以及对局部用于管程的数据进行初始化的部分。显然管程就是将描述临界资源的数据结构以及对该数据结构进行操作的一组互斥过程封装在一起所形成的一个类,但其中的数据结构和过程均为私有的,即局部于管程的数据结构仅能被管程内的过程所访问,而管程内的过程也仅能访问管程内的数据结构。当有多个进程都想调用管程中的过程以访问该管程所管理的临界资源时,编译程序就会检查该管程中是否有过程处于活跃状态,若有则将该请求进程阻塞,直到进入管程的进程离开时才唤醒被阻塞的进程。这种临界区的互斥进入是由编译程序自动完成的,无须程序员安排。管程机制的典型形式如下:

```
monitor  管程名
{
    普通变量定义部分;
    条件变量部分;
    过程定义部分;
    普通变量和条件变量初始化部分;
};
```

下面仍以计算和打印问题来说明这种方法的具体实现。

设 Pc()为计算进程,用于完成计算工作,并将计算结果存入缓冲区。执行保存结果的操作之前需判断缓冲区空否,若缓冲区满,则等待,否则执行。

设 Pp()为打印进程,它从缓冲区中取数据,并将数据送到打印设备上打印。执行前需判断缓冲区是否为空,若缓冲区空,则等待,否则执行。

设计一个管程,名为 calculate_print,其具体实现如下:

```
monitor calculate_print
{
    semaphore Bufempty,Buffull;                    /* 定义两个条件变量 */
    integer buf;
    procedure put_data(integer x)
    {
        wait(Bufempty);
        buf=x;
        Bufempty=false;
        Signal(Buffull);
    }
    function get_data(result):integer
    {
```

```
            wait(Buffull);
            result=buf;
            Buffull=false;
            Signal(Buffull);
        }
        Bufempty=true;
        Buffull=false;
    }
```

有两个信号量：Bufempty 和 Buffull。其中 Bufempty 表示缓冲区为空否，初值为 1，表示缓冲区为空；Buffull 表示缓冲区中是否有数据，初值为 0，表示缓冲区中无数据。

进程 Pc()和 Pp()的具体描述如下：

```
Pc()                                          /* 计算进程 Pc */
{
    while(true){
        result←Calculate();
        calculate_print.put_data(result)
    }
}
Pp()                                          /* 打印进程 Pp */
{
    while(true){
        calculate_print.get_data(result);
        print(result);
    }
}
```

8.3 经典的进程同步互斥问题

8.3.1 生产者和消费者问题

生产者和消费者问题及其同步技术是由 Dijkstra 于 1968 年提出的。当把并发进程的同步和互斥问题抽象为一般模型后，计算机系统中的许多问题都可归结为生产者和消费者问题，所以生产者和消费者问题的研究是非常有意义的。

一个进程如果使用某种资源（如缓冲区），则称该进程为消费者；一个进程如果释放这种资源，则称该进程为生产者。当把并发进程的同步和互斥问题一般化时，就可以得到一个抽象的模型，即生产者和消费者问题。

对于长度为 n 的有界缓冲区以及一群生产者 P_1、P_2、…、P_n 和一群消费者 C_1、C_2、…、C_n，每个生产者 P_i 生产消息数据，并把它放入有界缓冲区，消费者 C_i 从有界缓冲区中取出消息并处理它，如图 8.3 所示。生产者使用过程 producers(data)生产消息，消费者使用过程 consumers(data)从缓冲区中取出数据并进行处理。为了更好地对缓冲区进行管理，将有界缓冲区单元链成一个环形队列，并通过两个指针 Pf 和 Pe 分别指出满缓冲区和空缓冲区的头部单元，如图 8.4 所示。图中阴影部分为填入数据的满缓冲区单元，指针 Pf 和 Pe 顺时

针移动。



图 8.3　具有 n 个缓冲区的生产者和消费者问题

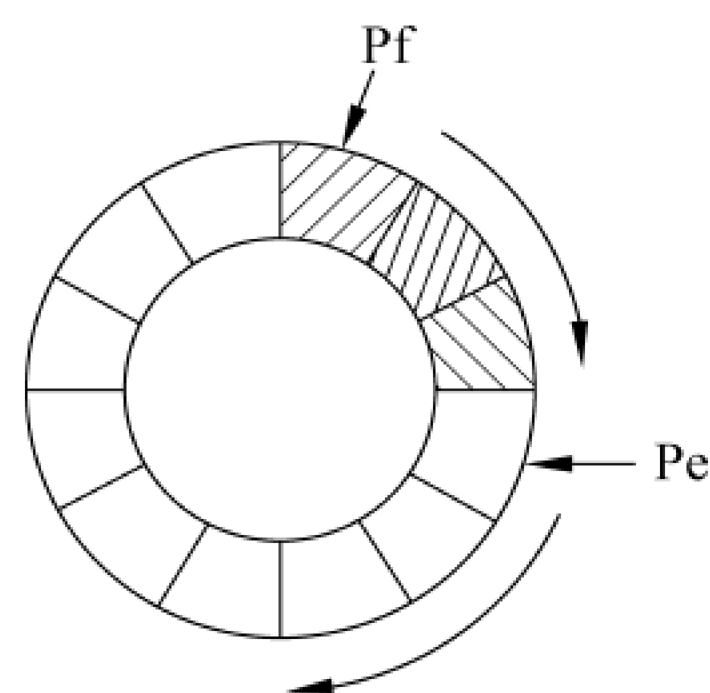


图 8.4　环形缓冲区

首先,生产者和消费者问题为一个同步问题,即生产者和消费者满足如下条件。

生产者:使用缓冲区资源,将数据存入缓冲区。执行上述操作的前提条件是缓冲区中至少有一个存储单元为空。

消费者:释放缓冲区资源,消费者从缓冲区中取出数据。执行上述操作的前提条件是缓冲区中至少一个存储单元内有数据。

此外,由于有界缓冲区为临界资源,因此各生产者进程和各消费者进程间必须互斥执行。

设有如下信号量:

Sem——公用信号量,初值为1,表示是否有进程在使用缓冲区,保证生产者进程和消费者进程对缓冲区的互斥操作。

Empty——生产者私有信号量,表示缓冲区中的空存储单元数,初值为 n;当 Empty=0 时,表示缓冲区中没有空的存储单元,此时生产者进程阻塞。

Full——消费者私有信号量,表示缓冲区中的非空存储单元数,初值为0;当 Full=0,表示缓冲区中存储单元均无数据,此时消费者进程阻塞。

```
#define MAXSIZE n;
typedef struct buffer {
     int data;
     struct buffer * next;
}  buf[MAXSIZE];                    /* buf[] is the structure array of the ring buffer */
semaphone Sem=1,Empty=MAXSIZE, Full=0;
producer(i,data)                    //生产者进程
{
     P(Empty);
     P(Sem);
     buf(Pe).data=data;
     Pe=buf[Pe].next;
     V(Full);
     V(Sem);
}
consumer(i,data)                    //消费者进程
{
```

```
    P(Full)
    P(Sem)
    data=buf(Pf).data;
    Pf=buf[Pf].next;
    V(Empty);
    V(Sem);
}
```

仅有一个生产者和一个消费者问题的管程解决方法如下：

```
monitor ProducerConsumer{
    condition full,empty;
    integer count;
    procedure insert(integer item)
    {
        if count==MAXSIZE then wait(empty);
        insert_item(item);
        count=count+1;
        if count==1 then signal(full);
    }
    function remove(item):integer
    {
        if count==0 then wait(full);
        item=remove_item;
        count=count-1;
        if count==MAXSIZE - 1 then signal(empty);
    }
    count=0;
}
Producer                        //生产者进程
{
    while(true){
        item=produce_item;
        ProducerConsumer.insert(item);
    }
}
Consumer                        //消费者进程
{
    while( true ){
        ProducerConsumer.remove(itme);
        consume_item(item);
    }
}
```

8.3.2 哲学家进餐问题

哲学家进餐问题涉及5个哲学家，他们交替地进行思考和进餐。他们分别坐在位于一

个圆形餐桌周围的5把椅子上,圆桌中央是一碗米饭,餐桌上共有5根筷子,分别摆放在每两个相邻座位的中间,如图8.5所示。当哲学家思考时,他不与其他人交谈。当哲学家饥饿时,他将拿起和他相邻的两根筷子进餐。但他很可能仅拿到一根,此时旁边的另一根正在他邻居的手中。只有他拿到两根筷子时才能开始进餐。完成进餐后,他将两根筷子分别放回原位,然后再次开始思考。这就是哲学家进餐问题。

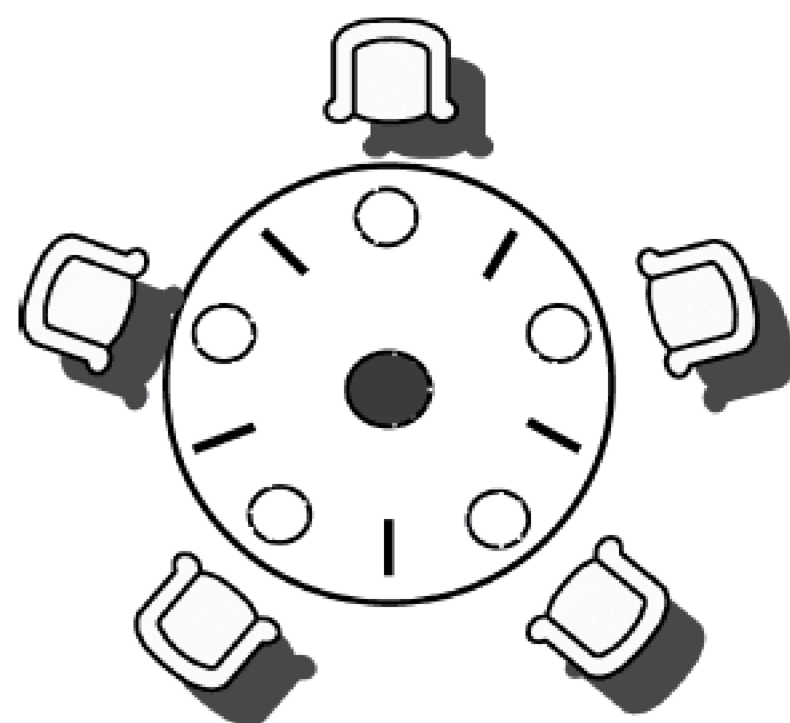


图8.5 哲学家进餐问题

下面采用信号量来解决上述问题。

一个简单的途径是每根筷子分别用一个信号量来表示。一个哲学家拿起一根筷子时,他需要在这根筷子所对应的信号量上完成P操作;同样,当他放下一个筷子时,需要在这根筷子所对应的信号量上完成V操作。通过数组chopstick[5]来表示与5根筷子所对应的信号量;这些信号量均被初始化为1。哲学家描述如下:

```
semaphone chopstick[5]={1,1,1,1,1};
main()
{
    cobegin
        Philosophers(0); Philosophers(1); Philosophers(2)
        Philosophers(3); Philosophers(4)
    coend
}
Philosophers(int i)                    //第 i个哲学家进程
{
    while(true)   {
        P(chopstick[i]);               //get the left chopstick
        P(chopstick[(i+1) %5]) ;       //get the right chopstick
                :
                :
        eating();
                :
                :
        V(chopstick[i]);               //release the left chopstick
        V(chopstick[(i+1) %5]);        //release the right chopstick
                :
                :
        thinking()
    }
}
```

上述解法是正确的,能够保证两个哲学家同时拿同一子筷子的情况不会发生;但有一个存在着问题:第$i(i=0,1,2,3,4)$个哲学家拿到了第i根筷子,又都申请第$i+1$根筷子,又都申请不到第$i+1$根筷子,即出现五个哲学家各拿到一根筷子又都等待被另一哲学家拿到的另外一根筷子的情况,这种情况即为死锁(deadlock),在8.5节讨论。

下面是哲学家进餐问题的管程解决方法：

```
enum status{eating,hungry,thinking};
monitor philosophers
{
    status state[N];
    condition self[N];
    test(int i)
    {
        if((state[(i-1) mod N]!=eating)&&(state[i]==hungry)&&(state[(i+1) mod
        N]!=eating))
        {
            state[i]=eating;
            self[i].signal;
        }
    };
    public:
    pickup(int i)
    {
        state[i]=hungry;
        test(i);
        if(state[i]!=eating) self[i].wait;
    };
    putdown(int i)
    {
        state[i]=thinking;
        test((i-1) mod N);
        test((i+1) mod N);
    };
    philosophers()
    {
        for(int i=0;i<N;i++) state[i]=thinking;
    };
};
```

设第 i 个哲学家进程为 thinker(int i)，其实现方式如下：

```
thinker(int i)                          //第 i 个哲学家进程
{
    while (true)
    {
        thinking;
        pickup(i);
        eating;
        putdown(i);
    }
}
```

8.3.3 读者和写者问题

一个数据集可以被多个并发进程所共享。一些进程也许仅对该数据集进行读取操作，而其他一些进程可能要更新(先读后写)该数据集的内容。为了区分上述两类进程，分别将它们称为读者和写者。很明显，当两个读者同时访问共享的该数据集时不会引起错误，但当一个写者和其他进程(无论是读者还是写者)同时访问该数据集时，可能就会造成混乱。为了确保上述问题不会出现，要求写者与写者、写者与读者之间必须互斥地访问共享数据。这种需求就引出了读者和写者问题。

下面用 P、V 原语描述读者和写者问题。

设共享变量 count 表示读者数量，初值为 0；对于读者在使用该变量时要互斥，设信号量 SC 对共享变量 count 实现互斥使用，其值为 1 时表示可以使用，初值为 1；信号量 SP 表示是读/写或写/写互斥，其值为 1 时允许读或写，初值为 1。读者-写者描述如下：

```
int count=0;
semaphone SC=1;
semaphone SP=1;
main()
{
    cobegin
        reader();
        writer();
    coend
}
reader()                              //读者进程
{
    while(true){
        P(SC);                        //申请使用 count 变量
        if count==0 then P(SP);       //申请进行读操作
        count=count+1;
        V(SC);                        //释放 count 变量
    进行读;
        P(SC);
        count=count-1;
        if count==0 then V(SP);       //释放读操作
        V(SC);
    }
}
writer()                              //写者进程
{
    while(true){
        P(SP);                        //申请进行写操作
        进行写;
        V(SP);                        //释放写操作
    }
}
```

在上述解法中,如果读者正在读,又有读者不断地到达,则可能出现写者无限期地等待的情况发生,这是不合理的,因为写者操作是更新数据的,读者要想读到最新数据,写者应当优先进行。上述解法通常称为读者优先的读者和写者问题(或称偏向读者的读者和写者问题)。下面给出写者优先的读者和写者问题(或称偏向写者的读者和写者问题)。

为了保证写进程及时进行数据更新,应允许正在读数据的读进程继续执行直到完成读操作,而不允许后续读进程进行读操作,为此设一个信号量 SW 表示是写进程申请写操作,其值为 1 时没有写进程申请写操作,初值为 1;共享变量 count 表示读者数量,初值为 0;对于读者在使用该变量时要互斥,设信号量 SC 对共享变量 count 实现互斥使用,其值为 1 时表示可以使用,初值为 1;信号量 SP 表示是读/写或写/写互斥,其值为 1 时允许读或写,初值为 1;读者和写者描述如下:

```
int count=0;
semaphone SC=1;
semaphone SP=1;
semaphone SW=1;
main()
{
    cobegin
        reader();
        writer();
    coend
}
reader()                            //读者进程
{
    while(true){
        P(SW);                      //是否有写进程申请写操作
        P(SC);
        if count==0 then P(SP);
        count=count+1;
        V(SC);
        V(SW);
        进行读;
        P(SC);
        count=count-1;
        if count==0 then V(SP);
        V(SC);
    }
}
writer()                            //写者进程
{
    while(true){
        P(SW);                      //是否有写进程申请写操作
        P(SP);
        进行写;
```

```
            V(SP)
            V(SW);
        }
}
```

8.3.4 理发师睡觉问题

理发店里有一位理发师、一把理发椅和5把(这里假设为5把)供等候理发的顾客坐的椅子。如果没有顾客,则理发师便在理发椅上睡觉,如图8.6所示。当一个顾客到来时,他必须先叫醒理发师,如果理发师正在理发时又有顾客到来,则如果有空椅子可坐,顾客就坐下来等。如果没有空椅子,顾客就离开。这里的问题是为理发师和顾客各编写一段程序来描述他们的行为,要求不能带有竞争条件。

对理发师睡觉问题,理发师和顾客的工作流程如图8.7所示。

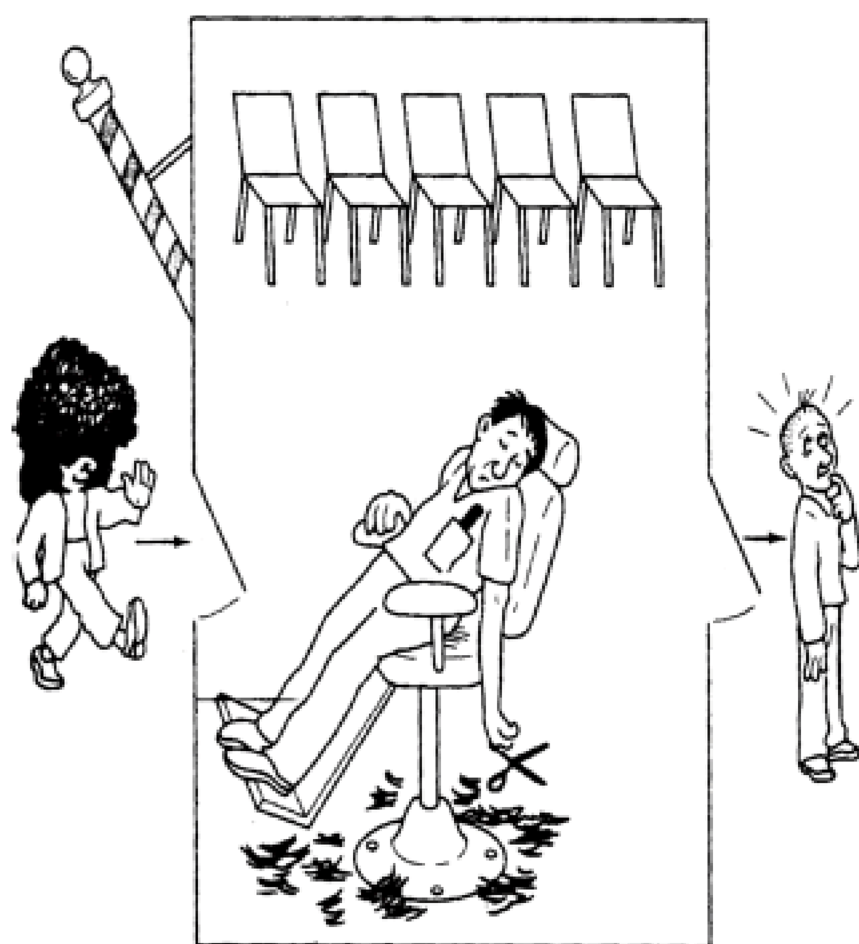


图8.6 理发店

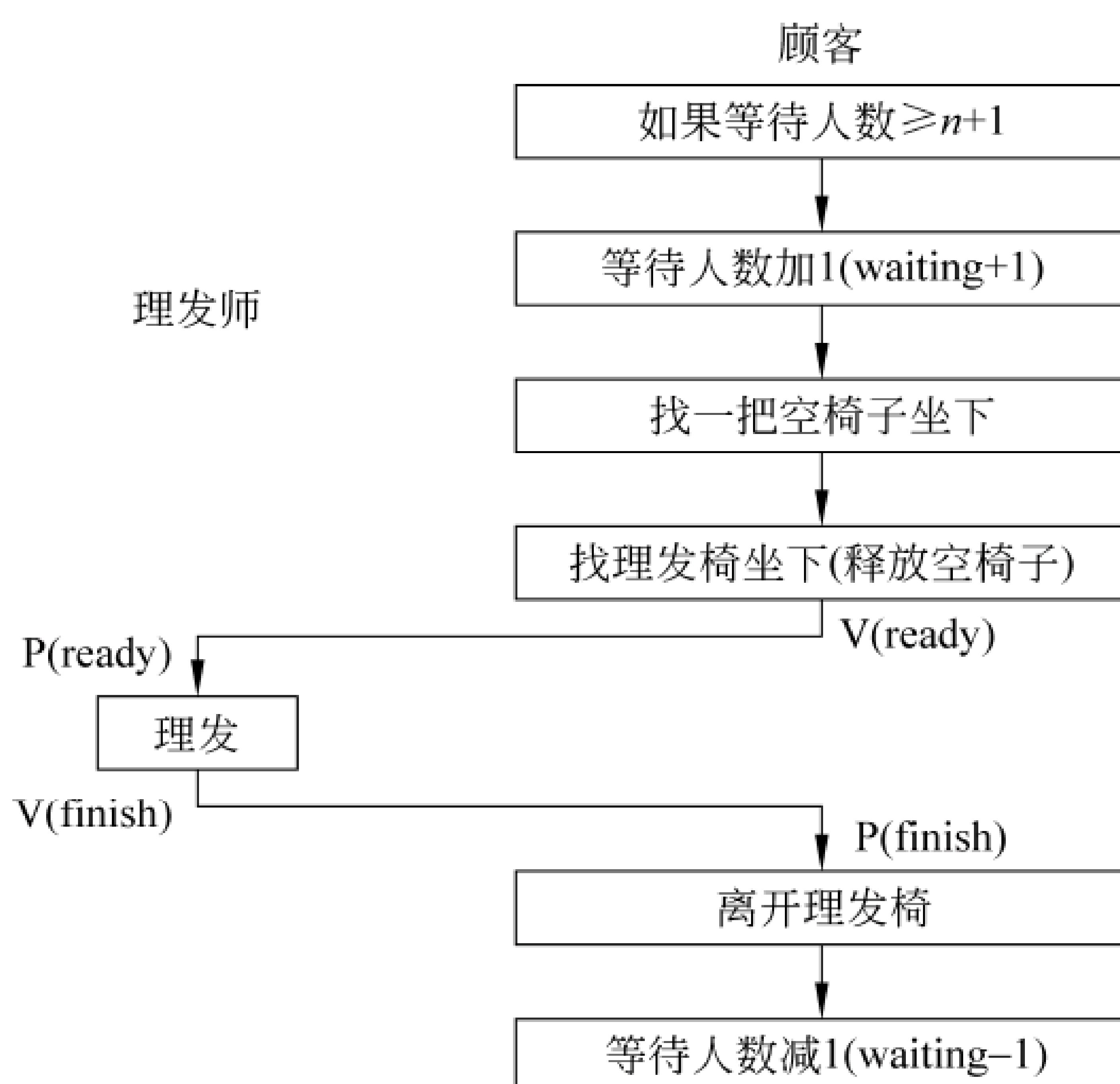


图8.7 理发师和顾客的工作流程

第一步需要考虑等待的顾客数(即坐在椅子上的顾客数),设置一个变量waiting(初值为0),当进来一个顾客时,waiting增1,当一个顾客理发时,waiting减1。

第二步需要考虑对waiting共享变量的互斥操作,设置一个信号量mutex(初值为1),完成对waiting共享变量的互斥操作。

第三步需要考虑空椅子个数,每把空椅子是一个临界资源,设置一个信号量wchair(初值为5)。

第四步考虑是否有顾客坐在理发椅上,理发椅是一个临界资源,设置一个信号量bchair(初值为1)。

第五步为顾客和理发师的同步操作,设置ready和finish两个信号量(初值均为0),前者表示顾客是否准备好,后者表示理发师是否完成理发。

理发师睡觉问题的算法可描述如下:

```
int waiting=0;                    //等待的顾客(含正在理发的人数,最多不超过6人)
```

```
semaphore mutex=1;              //用于 waiting 变量的互斥操作
semaphore bchair=1;             //理发椅的个数
semaphore wchair=5;             //空椅子的个数
semaphore ready=0;              //是否有顾客准备好
semaphore finish=0;             //理发师是否完成理发
main()
{
    cobegin
        barber();
        customer();
    coend
}

void barber()                   //理发师进程
{
    while(true)
    {
        P(ready);
        理发;
        V(finsish);
    }
}
void customer()                 //顾客进程
{
    P(mutex);
    if(waiting<6)
    {
        waiting=waiting+1;
        V(mutex);
    }
    else
    {
        V(mutex);
        离开;
    }
    P(wchair);
    P(bchair);
    V(wchair);
    V(ready);
    坐着理发;
    P(finish);
    V(bchair);
    P(mutex);
    waiting=waiting-1;
    V(mutex);
}
```

8.4 进程通信

在一个计算机系统内执行的并发进程可能是相互独立的一组进程，也可能是一组协作进程。如果一个进程在不影响其他进程的同时也不被其他进程所影响，则认为它是独立的；而一个进程影响系统内运行的其他进程或被系统内运行的其他进程所影响，则认为它是一个协作进程。很明显，和其他进程共享数据的进程为协作进程，协作进程间要进行必要的信息交换，即进行进程通信，以协作完成共同的任务。生产者和消费者问题是协作进程的最好实例。显然，进程通信（InterProcess Communication，IPC）就是在进程间交换信息。进程的同步和互斥实质上也可归结为进程通信，只不过一般仅传送几个字节，主要用于控制进程的执行速度，故常将进程的同步和互斥称为低级通信。在进程互斥中，进程通过只修改信号量来向其他进程表明临界资源是否可用。在生产者-消费者问题中，生产者通过缓冲池将所生产的产品传送给消费者。

应当指出，信号量机制作为同步工具是卓有成效的，但作为通信工具则不够理想，主要表现在下述两方面：

（1）效率低。生产者每次只能向缓冲池投放一个产品（消息），消费者每次只能从缓冲区中取得一个消息。

（2）通信对用户不透明。

可见，用户要利用低级通信工具实现进程通信是非常不方便的。因为共享数据结构的设置、数据的传送以及进程的互斥与同步等都必须由程序员去实现，操作系统只能提供共享存储器。

本节所要介绍的是高级进程通信，是指用户可直接利用操作系统所提供的一组通信命令高效地传送大量数据的一种通信方式。这些通信命令对于用户来讲是透明的，用户不必了解这些命令的具体实现细节，从而减轻了用户的编程负担。

8.4.1 进程通信的类型

目前，计算机系统中比较普遍采用的进程通信方式有 3 种：共享存储器、消息传递系统和管道通信。

1. 共享存储器

在共享存储器系统（Shared-Memory System）中，相互通信的进程共享某些数据结构或共享存储区，进程之间能够通过这些空间进行通信。据此，又可把它们分成以下两种类型：

（1）基于共享数据结构的通信方式。在这种通信方式中，要求诸进程公用某些数据结构，借以实现诸进程间的信息交换。如在生产者-消费者问题中，就是用有界缓冲区这种数据结构来实现通信的。这里，公用数据结构的设置及对进程间同步的处理都是程序员的职责。这无疑增加了程序员的负担，而操作系统却只需提供共享存储器。因此，这种通信方式是低效的，只适于传递相对少量的数据。

（2）基于共享存储区的通信方式。为了传输大量数据，在存储器中划出了一块共享存储区，诸进程可通过对共享存储区中数据的读或写来实现通信。这种通信方式属于高级通信。进程在通信前，先向系统申请获得共享存储区中的一个分区，并指定该分区的关键字；

若系统已经给其他进程分配了这样的分区,则将该分区的描述符返回给申请者,继之,有申请者把获得的共享存储分区连接到本进程上;此后,便可像读、写普通存储器一样地读、写该公用存储分区。

2. 消息传递系统

不论是单机系统、多机系统还是计算机网络,消息传递机制都是广泛采用的一种进程通信机制。在消息传递系统中,进程间的数据交换以具有一定格式的消息(message)为基本单位,该消息在计算机网络中被称为报文。系统为用户提供了一组通信原语用于实现进程间的消息传递,如发送原语和接收原语等。消息传递系统根据实现方式的差异可分为两类:

(1) 直接通信方式,简称消息缓冲通信。发送进程将消息直接发送给接收进程,由操作系统将该消息挂在接收进程的消息缓冲队列上供接收进程读取。

(2) 间接通信方式,简称信箱通信。发送进程将消息发送到接收进程的信箱中供接收进程读取。这种方式广泛应用于计算机网络中。这种通信方式较为简单,由于信箱可以位于主存中,所以数据传输的速度较高。

3. 管道通信

所谓管道是指用于连接一个读进程和一个写进程以实现它们之间通信功能的一个共享文件,又称为 pipe 文件。它是由 UNIX 系统首先提出的一种进程通信方式。向管道(共享文件)提供输入的发送进程(即写进程)以字符流形式将大量数据写入管道,而接收进程(即读进程)则接收管道输出的数据。由于发送进程和接收进程是利用管道进行通信的,故称为管道通信。

为了协调发送进程和接收进程间的通信,管道通信机制必须提供如下三方面功能:

(1) 互斥。当一个进程在对 pipe 文件进行读/写操作时,另一个进程必须等待。

(2) 同步。当写进程将要传递的数据写入管道后就唤醒读进程,然后便进入睡眠等待状态,直到读进程读走数据后,再把它唤醒。当读进程读空管道后就转去唤醒写进程,然后转入睡眠等待状态,直到写进程将新数据写入管道,再将它唤醒。

(3) 确认对方是否存在。只有确认对方已存在时才能进行管道通信,否则会造成因对方不存在而无休止地等待。

管道通信方式具有如下特点:

(1) 管道以文件为中间传输手段,因此可以进行大批量的数据传输。

(2) 管道以字符流形式写入和读出,不必以消息为单位。

(3) 管道以 FIFO 方式工作,先写入的数据先被读出。

以上 3 种进程通信方式都可以实现大量数据的传输,但通信方式不同,需使用不同的控制方式实现进程间的同步和互斥。下面介绍常用的消息传递机制,而管道通信方式在 8.7.1 节详细介绍。

8.4.2 消息传递通信

消息传递通信是进程间经常采用的通信方式,下面分别介绍两种类型的消息传递系统。

1. 直接通信方式

直接通信是指发送进程利用操作系统所提供的发送命令直接将消息传递给目标进程。消息缓冲属于直接通信,它是由 Hansen 在 1973 年提出的一种进程通信方式。在这种方式

下，发送进程利用操作系统提供的发送原语 send(Receiver，message)将消息直接发送给目标进程，而接收进程则利用操作系统提供的接收原语 receive(Sender，message)接收发送者发送给它的消息。所以，在直接通信方式中，要求发送方和接收方必须显式地提供对方的标识信息。系统所提供的两条通信原语如下：

（1）发送原语 send(Receiver，message)。将由 message 所指定的消息发送给由 Receiver 所指定的接收进程。

（2）接收原语 receive(Sender，message)。接收由 Sender 所标识的发送进程所发送来的消息 message。

在某些情况下，接收进程可能与多个进程进行通信。所以此时无法指定发送进程。例如，打印服务进程可以接收多个用户或进程发来的打印服务请求。对于这种情况，接收原语中用于标识发送进程的参数用于存储完成通信后的返回值，此时接收原语可表示为

```
receive(id,message)
```

下面利用消息缓冲机制来描述生产者和消费者问题。当生产者进程产生一个消息后，它便调用 send()原语将消息发送给消费者进程，而消费者进程则利用接收原语 receive()来接收这个消息。如果没有消息产生，则消费者进程必须等待，直到生产者进程将产生的消息发送过来。生产者和消费者问题描述如下：

```
producer()
{
    repeat
        ⋮
        produce an data to message;
        ⋮
        send(consumer,message)
    until false
}
consumer()
{
    repeat
        receive(producer,message)
            ⋮
        consume an data from message;
    until false
}
```

2. 间接通信方式

间接通信方式是指进程间的通信需要一个作为缓冲区的数据结构实体。该实体用来暂存发送进程发送给目标进程的消息，而接收进程则从该实体中取出发送给它的消息，故该中间实体也称为信箱(mail box)。该信箱可以安全地保存信息，它既可以实现实时通信，也可以实现非实时通信。

系统为信箱通信提供了若干条原语，分别用于信箱的创建、撤销以及消息的发送、接收等。

1）创建和撤销原语

信箱创建原语用于建立一个新信箱。调用创建原语时需给出信箱名字、信箱属性（公有、私有或共享）以及共享者等必要信息。当进程不再需要信箱时可调用撤销原语将之撤销。

2）消息的发送和接收原语

当进程使用公用信箱进行通信时，需调用系统提供的通信原语进行通信。系统所提供的通信原语如下：

（1）发送原语 send(mailbox A，message M)：将消息 M 发送到 A 所标识的信箱。

（2）接收原语 receive(mailbox A，message N)：从 A 所指定的信箱中接收一个消息到 N。

通信过程中所使用的信箱可由操作系统创建，也可由用户进程创建，创建者为信箱的拥有者。发送到信箱中的消息不但表示所要交换的大量数据，还表示两个相互通信的进程间的地位是平等的。

信箱类型的一般定义如下：

```
typedef mailbox struct{
    int in,out;
    semaphone s1,s2;
    semaphone mutex;
    message letter[k];
};
```

其中，in 和 out 分别为读、写指针，初值为 0，取值范围为 $0\sim k-1$；s1 和 s2 为协调发送与接收同步的信号量，其初始值分别为 k 和 0；mutex 用于对信箱的互斥操作，初值为 1。这些信息构成信箱的头。letter 为信箱体，可以保存 k 封信。

发送和接收进程定义如下：

```
send(mailbox A ,message M)
{
    P(A.s1);
    P(A.mutex);
    A.letter[A.in]=M;
    A.in=(A.in+1)%A.k;
    V(A.mutex);
    V(A.s2);
}
```

```
receive (mailbox A ,message N)
{
    P(A.s2);
    P(A.mutex);
    N=A.letter[A.out];
    A.out=(A.out+1)%A.k;
    V(A.mutex);
    V(A.s1);
}
```

8.5 死　　锁

在计算机系统中，资源是有限的，而且许多资源是独占资源（如打印机），在任何时刻它只能被一个进程所使用。另一方面，一个进程在其整个生命周期内一般不可能仅仅使用一种资源，它可能申请使用多种资源。并发性是现代操作系统的基本特征，而资源共享是进程

并发执行的必要条件。资源共享和进程的并发执行必然导致对有限系统资源的激烈竞争。当一个进程申请一个资源而该资源目前得不到满足时，它将进入等待状态，而此时该进程所申请的资源可能正被另一个处于等待状态的进程所占有，这种因等待进程所请求分配的资源被对方进程所相互占有，从而导致两个或多个处于等待状态的进程永远不能改变其状态的现象，就称为死锁（deadlock）。

死锁是现代操作系统所必须解决但尚没有得到根本解决的问题。本节介绍死锁的基本概念，分析死锁产生的原因，提出避免死锁发生的方法，以及当死锁发生时如何检测和恢复系统。

8.5.1　死锁的基本概念

死锁是指一个进程集合，这些进程处于永久阻塞状态，它们正在竞争有限的系统资源或正在进行相互通信。此时，集合中的各个进程都在等待某些事件的发生，而这些事件又仅仅需要被该集合中的另外一些进程所触发，从而没有一个进程能够继续运行下去，集合中的所有进程都处于永远相互等待状态。如系统中存在两个进程 P_1 和 P_2，它们分别占有系统资源 S_1 和 S_2，但分别需要获得 S_2 和 S_1 后才能运行，如图 8.8 所示，此时系统即发生了死锁。

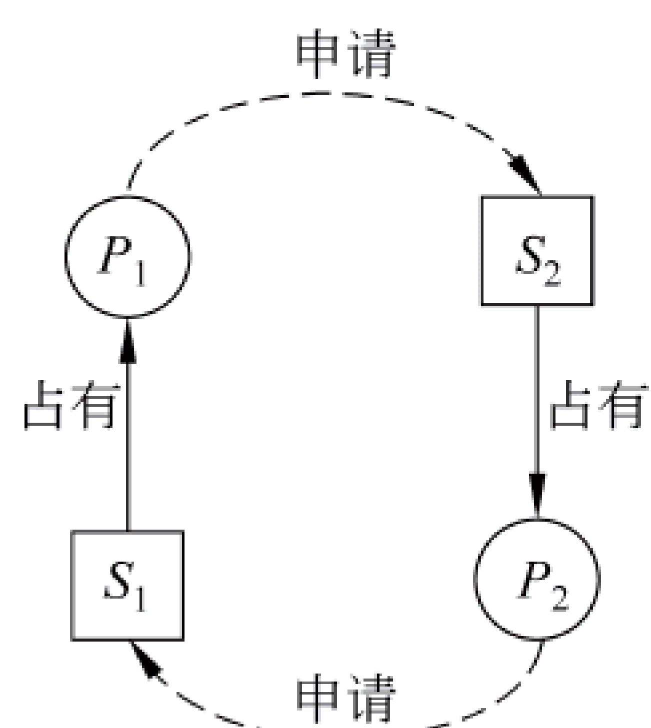


图 8.8　两个进程的死锁模型

产生死锁的原因可以归结为两点：有限资源的竞争和并发进程推进的顺序非法。当系统中供多个并发进程共享的资源不能同时满足各个进程执行的需要时，如果此时进程的推进顺序非法，就会因各个进程争夺资源而造成系统死锁。竞争的系统资源可能是硬件（如存储器、I/O 设备等），也可能是一组信息，但它们均为临界资源，每次仅能为一个进程所使用。

死锁起因于并发进程对共享资源的竞争。当并发进程所需的资源个数多于系统所能提供的资源数时，系统就可能发生死锁。系统资源可分成两类：可剥夺资源（如存储器）和不可剥夺资源（如打印机）。一个进程可以将可剥夺资源从占有它的进程中抢夺过来，而不会引起任何问题；但对于不可剥夺资源则行不通。竞争临界资源所造成的死锁和资源的不可剥夺性存在一定的关系，但并不是导致死锁发生的直接原因，可以采用适当的资源分配算法来避免死锁现象的发生。造成死锁现象发生的主要原因是并发进程执行的不确定性和随机性。当多个并发进程同时执行时，各进程推进顺序的不当会直接导致死锁现象的发生。

在多道程序环境下，各并发进程的执行可能相当顺利，也可能在其执行过程中发生死锁。那么在什么条件下会发生死锁呢？下面给出死锁发生的 4 个必要条件。

（1）互斥条件。共享资源的使用是互斥的，某一段时间内共享资源只能被一个进程占用。如果此时其他进程也要求使用该资源，申请者必须阻塞，直到占有该资源的进程释放该资源后，它才能被唤醒。

（2）不可剥夺条件。进程所使用的共享资源在未用完之前，不能由其他进程强行剥夺，只能自己释放。

（3）部分分配。进程在申请新资源的同时，将继续占有已获得的资源，而申请的新资源此时却被其他进程所占有。

(4) 环路条件。存在进程-资源循环链,链中每一个进程已占用的资源同时被下一个进程所申请。

在有些操作系统的论述中,出现一个与死锁相关的名词——饥饿(starvation)。饥饿经常发生在系统资源的动态分配过程中,虽然系统使用了一些显得十分合理的资源分配策略,但由于资源动态分配的复杂性以及进程的并发性,致使一些进程可能永远申请不到所需要的资源,但此时系统并不发生死锁,仍存在处于运行状态的进程,这时就称申请不到所需资源的这些进程处于饥饿状态。饥饿与死锁间存在本质性的区别,处于饥饿状态的进程可能是一个,也可能存在多个,而处于死锁状态的进程一定是两个以上;饥饿不是对系统资源的循环等待,而是单向等待;处于饥饿状态的进程所等待的事件不是永远不会发生,而是发生时总被其他进程抢先响应。饥饿现象是可以避免的,例如采用先来先服务资源分配策略就可以避免进程长时间等待分配资源现象的发生,此时进入系统的进程只要等待足够长的时间一定能够得到所需的资源。

8.5.2 死锁的解决方案和方法

1. 解决死锁的方案

死锁造成的危害有可能是巨大的,系统可能会因此而崩溃。现代操作系统中都引入了很多有效的解决方法。下面就借助于模型和实例介绍各种解决死锁的方案。

1) 鸵鸟政策

这是一种最简单的方法,就像鸵鸟把头埋进沙子,对危险视而不见一样,系统对死锁不加理会。

鸵鸟政策的出发点是,由于并发系统中的死锁现象并不是每时每刻都会发生,在早期的系统中是极偶然的现象,所以,系统的设计者宁愿花更多的精力和资源解决更加棘手、出现更为频繁的问题。

2) 不让死锁发生

这是一种以遏制死锁为出发点的想法,即在进程执行前和执行过程当中,对资源的分配加以限制,使系统永远不会死锁。对资源分配的策略可以分为静态策略和动态策略。

静态策略是指进程在创建时就由系统为其分配了所有资源,满足后方可执行,并且在以后的执行过程中没有资源分配工作。这种策略防止死锁当然很有效,而且易于实现,但对资源造成了不必要的浪费。因为所有进程都要满足最大资源数才能执行,并发的进程数量肯定要少,系统中的某些资源可能使用的时间很短,却被一个进程从头到尾占用,而其他想要使用这个资源的进程却得不到满足,增大了等待时间,系统效率极低。

动态策略相对静态策略有更高的灵活性,它可以在进程执行的时候改变资源的分配情况,避免静态策略的呆板,虽然实现技术相对复杂,但系统效率会提高很多,资源利用率也相对较高。关于动态策略的具体实现在以后的内容中有更详细的说明。

3) 让死锁发生

如果说不让死锁发生是事前控制,那么让死锁发生就是事后弥补的方法,主要是指当死锁发生时采取的应对措施。毕竟用户使用计算机时更注重效率,而不让死锁发生的策略都有些极端,当死锁偶然发生的时候再来解决也为时不晚。

2. 解决死锁的方法

针对上述死锁的解决方案,提出的死锁的解决方法一般分为预防、避免、检测与恢复3种。

死锁的预防一般是从破坏导致死锁发生的必要条件着手,采用某种策略,限制并发进程对资源的请求,使得死锁的必要条件在系统运行期间得不到满足,从而预防死锁现象的发生。

避免是指系统在为进程动态分配资源时,根据系统资源的使用情况,通过一定的算法提前对系统状态做出预测,避免死锁的发生,也称为动态预防,即在资源的动态分配过程中预测出死锁发生的可能性并加以避免。

检测与恢复是死锁发生后的事后处理技术。它是指系统设有专门的机构,当死锁发生时该机构能够检测到死锁发生的位置和原因,并能通过外力破坏死锁发生的必要条件,使得并发进程从死锁状态中恢复出来。

由于操作系统的并行性、共享性以及随机性等特点,很难通过预防和避免方法达到排除死锁的目的,故一般都采用检测与恢复的方法。

8.5.3 死锁的预防

产生死锁的必要条件已经了解了,预防死锁的工作就是从破坏这些条件入手。

1. 破坏互斥条件

我们知道,允许两个进程同时使用打印机这种独占资源会造成混乱,如果资源允许进程共享,那么死锁肯定不会产生。通过借助SPOOLing技术可以允许若干个进程同时产生打印数据。该方法中唯一真正申请物理打印机的进程是打印精灵进程,由于它不会申请别的资源,所以不会因打印机产生死锁。但SPOOLing技术并不适用于所有的资源,如进程表。

破坏资源的"互斥性"比较困难,因为这种方法对大多数资源行不通,所以只能看其他3个条件。

2. 破坏"不可剥夺"条件

允许进程剥夺也应当包括剥夺自己的"请求",也就是进程申请资源得不到满足时,进程可以收回请求,转去做其他的工作。

有两种可以破坏这个条件的方法。

一种方法是,如果一个已经占有资源的进程再次申请资源时,所申请的资源不能得到满足,它必须先放弃已经占有的资源,若这些资源还没有使用完,则只能以后一起申请。

另一种方法只适用于申请资源的进程优先级比占有该资源的进程优先级高的情况,如果一个进程申请的资源正被别的进程占用,若申请进程的优先级高,它就可以强迫正占有资源的进程放弃使用。

这些方法实现起来相当困难,为了保存进程放弃资源的现场以及以后的再次恢复,往往要耗费很多时间和存储空间。一般来说,破坏"不可剥夺"条件只适合于处理机和存储器资源,对于其他资源,不宜使用此方法。

3. 破坏"部分分配"条件

许多操作系统破坏"部分分配"条件时都采用静态分配策略。静态分配是指当一个进程得到了它所需要的所有资源后才能执行。利用这种分配机制,进程在执行的过程中就不需

要申请资源了，死锁的 4 个必要条件之一的“部分分配”得以破坏。

这种方法的缺点是资源的利用率低，当进程创建时申请了所有要用到的资源，虽然省去了执行过程中的资源申请工作，但被占用的资源一经申请，不管在什么时间使用，直到进程结束才得以释放，资源利用率很低，系统效率也不高。

4. 破坏“环路”条件

按序分配资源的方法对破坏“环路”条件很有效。系统依据一定的策略给资源编号，例如按照资源特性、使用的频度或数量的多少等，由低到高顺序编号，进程必须按从小到大的顺序申请资源，并规定进程占有的资源号小于申请的资源号时才能得到申请资源。

例如，进程 A 占有 3 号资源，现在又申请 5 号资源，占有资源号小于申请资源号，此申请可以满足。进程 B 占有 5 号资源，现在又申请 3 号资源，5 大于 3，此申请不能满足，进程 B 要想得到 3 号资源，必须先放弃 5 号以及所有编号比 3 大的资源。

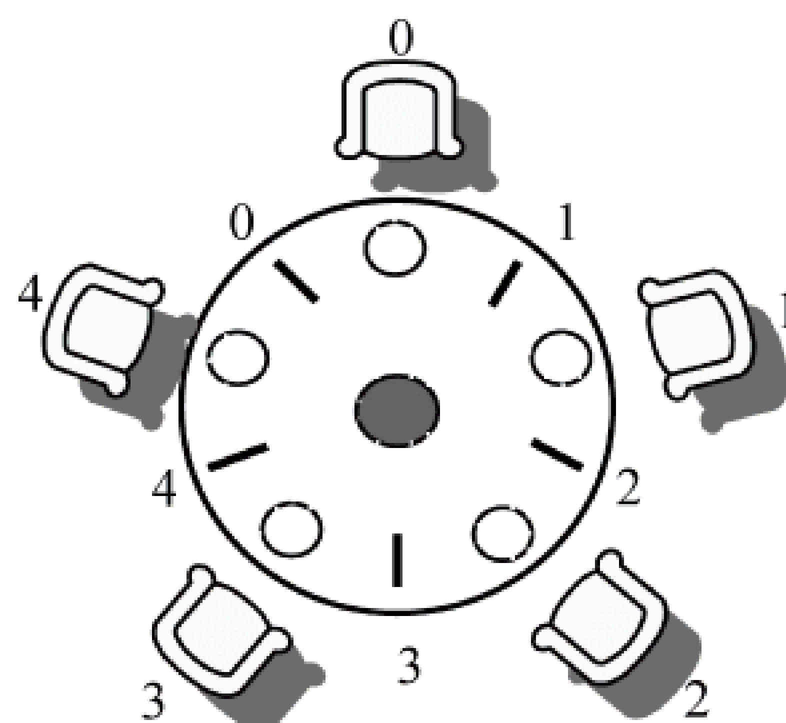


图 8.9 哲学家进餐问题

再来看那个著名的哲学家进餐问题。如图 8.9 所示，将 5 根筷子依次编号为 0～4，规定哲学家必须先拿小序号的筷子，再拿大序号的筷子。若小序号的筷子正被占用，他就进入阻塞，直到小序号的筷子被放下。这样，即使 5 个哲学家同时伸出左手，第 4 号哲学家应先拿第 0 号筷子，但第 0 号筷子被 0 号哲学家占据，所以，第 4 号哲学家因为不能拥有第 0 号筷子而无法申请第 4 号筷子，因而被阻塞。这样，拿第 3 号筷子的第 3 号哲学家同时可以拿到第 4 号筷子，先吃完了米饭，释放其占据的筷子，唤醒其他哲学家进程。依此类推，最终大家都可以顺利地吃完。

设 5 个哲学家对应 5 个进程 P_0，P_1，P_2，P_3，P_4，5 根筷子对应 5 个资源 r_0，r_1，r_2，r_3，r_4。进程 P_i 必须先申请筷子 r_i，再申请筷子 r_{i+1}，进程 P_4 必须先申请筷子 r_0，再申请筷子 r_4。设 5 个信号量为 S_0，S_1，S_2，S_3，S_4，初值为 1。5 个哲学家进餐的算法如下：

```
semaphone S0,S1,S2,S3,S4;
S0=S1=S2=S3=S4=1;
process Pi (i=0,1,2,3)
{
  while(true){
      thinking;
      hungry;
      P(Si);
      pickup  ri;
      P(Si+1);
      pickup  ri+1;
      eating;
      putdown  ri;
      putdown  ri+1
      V(Si);
      V(Si+1);
  }
```

```
};
process P4
{
  while(true){
       thinking;
       hungry;
       P(S0);
       pickup r0;
       P(S4);
       pickup r4;
       eating;
       putdown r0;
       putdown r4;
       V(S0);
       V(S4);
  }
};
```

对资源按序号分配的时候要注意:如果资源有多种类型,那么也要将这些资源类型按照一定的策略安排成一个序列,进程申请资源的时候先要确定类型的高低,再看此类型中各资源的序号大小。

8.5.4 死锁避免的方案——银行家算法

最具有代表性的死锁避免算法为由 Dijkstra 提出的银行家算法。该算法通过资源的试探分配来判断系统是否处于安全状态,以决定是否真正地进行资源分配,从而达到避免死锁发生的目的。系统的安全状态是指操作系统能够保证所有的进程在有限的时间内得到所需要的全部资源,否则说明系统是不安全的。处于不安全状态的系统可能导致死锁现象的发生,但只要保证系统处于安全状态,即可以避免系统死锁。

下面首先给出银行家算法中所用到的各种数据结构,然后描述算法实现的具体细节。

1. 银行家算法中的数据结构

(1) 可用资源向量 Available。该向量包括 m 个元素,其中的每个元素代表一类可以使用资源的数量,其初值是系统中所配置的该类资源的总数。向量中每个元素的值随着资源的分配和回收而动态变化。如果 Available[j]=k,则表示目前系统中有 k 个 R_j 类资源可供使用。

(2) 最大需求矩阵 Max。Max 为一个 $n \times m$ 维矩阵,它定义了系统中的每个进程(共 n 个进程)对 m 类资源的最大需求。如果 Max[i,j]=k,则表示进程 i 当前需要 k 个 R_j 类资源。

(3) 分配矩阵 Allocation。它也是一个 $n \times m$ 维矩阵,它定义了系统中的每个进程当前已获得的各种资源的数量。如果 Allocation[i,j]=k,则表示进程 i 目前已获得 k 个 R_j 类资源。

(4) 需求矩阵 Need。它也是一个 $n \times m$ 维矩阵,它定义了系统中的每个进程尚需申请

的各种资源的数量。如果 Need[i,j]=k，则表示进程 i 目前还需要申请 k 个 R_j 类资源。

显然，上述几个矩阵的关系如下：

$$Need[i,j]=Max[i,j]-Allocation[i,j]$$

2. 银行家算法的描述

设 Request$_i$[]是进程 P_i 的资源请求向量，如果 Request$_i$[j]=k，则表示进程 P_i 申请 k 个 R_j 类资源才能执行。当进程 P_i 提出资源请求后，系统执行如下过程：

(1) 如果 Request$_i$[j]≤Need[i,j]，则转向步骤(2)；否则认为出错，因为它申请的资源数超过了最大值。

(2) 如果 Request$_i$[j]≤Available[i,j]，则转向步骤(3)；否则表示系统内无足够的 R_j 类空闲资源，进程 P_i 申请的资源得不到满足，应将它挂在相应的等待队列中。

(3) 系统试探性地将资源分配给进程 P$_i$，并修改相应的数据结构：

$$Available[j]=Available[j]-Request_i[j]$$
$$Allocation[i,j]=Allocation[i,j]+Request_i[j]$$
$$Need[i,j]=Need[i,j]-Request_i[j]$$

(4) 系统执行安全检查算法，检查此次资源分配后系统是否处于安全状态。若安全，系统才正式将资源分配给进程 P_i，以完成本次分配；若不安全，系统将本次试探性资源分配作废，恢复原来的资源分配状态以及各种数据结构的值，阻塞进程 P_i。

3. 安全检查算法

判断系统安全性的检查算法的基本思想是：当一个进程对共享资源提出申请后，系统若满足其要求，则需要在其余的进程中寻找出一个执行序列，使得这个序列中的每个进程的最大资源请求均得到满足。若找到这样一个进程执行序列，则说明此次为进程分配资源是安全的。安全性检查算法具体描述如下：

(1) 设置两个临时向量：

① 工作向量 Work：它具有 m 个元素，表示系统可提供给进程继续运行所需要的各类资源数目，其初值为 Available。

② 二值向量 Finish：它表示进程是否已运行结束。开始时令 Finish[i]=false；当进程获得了足够资源并运行完成时，令 Finish[i]=true。

(2) 从进程集合中找到一个能满足下述条件的进程：

$$Finish[i]=false$$
$$Need[i,j]\leq Work[j]$$

若找到，则执行步骤(3)；否则执行步骤(4)。

(3) 当进程 P$_i$ 获得资源后，可顺利执行，直到完成，然后释放分配给它的资源，此时应执行

$$Work[j]=Work[j]+Allocation[i,j]$$
$$Finish[i]=true$$

转向步骤(2)。

(4) 如果所有进程的 Finish[i]=true 都满足，则表示系统处于安全状态；否则系统处于

不安全状态。

例8.3 设系统中存在5个进程{P_1,P_2,P_3,P_4,P_5}和3种共享资源{R_1,R_2,R_3},每种资源的数量分别为:$R_1=10$,$R_2=5$,$R_3=7$,假设在某一时刻资源的分配状态如表8.1所示。

表8.1 某时刻各进程的资源分配情况

进程	Max			Allocation			Need			Available		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	7	5	3	0	1	0	7	4	3	3	3	2
P_2	3	2	2	2	0	0	1	2	2			
P_3	9	0	2	3	0	2	6	0	0			
P_4	2	2	2	2	1	1	0	1	1			
P_5	4	3	3	0	0	2	4	3	1			

经分析此时系统存在一个安全序列{P_2,P_4,P_5,P_3,P_1},故目前的系统状态为安全状态。

现假设进程P_2进一步提出资源申请,当前资源请求向量为$\text{Request}_2=[1,0,1]$,即进程P_2请求分配1个资源R_1和1个资源R_3,下面分析满足进程P_2此次资源请求后,系统是否存在一个进程的安全执行序列。若存在,则说明此次为进程P_2分配资源后的系统状态是安全的,进程P_2可以获得资源并继续执行;若不存在,则说明此次为进程P_2分配资源后的系统状态是不安全的,系统不为进程P_2分配资源,应阻塞进程P_2。下面通过银行家算法完成上述功能,执行的具体步骤如下:

(1) 因为$\text{Request}_2\leqslant\text{Need}[2]$,即$[1,0,1]\leqslant[1,2,2]$,所以继续执行(2)。

(2) 因为$\text{Request}_2\leqslant\text{Available}$,即$[1,0,1]\leqslant[3,3,2]$,所以继续执行(3)。

(3) 对描述资源状态的各个变量进行修改:

```
Available=Available-Request2=[2,3,1]
Allocation[2]=Allocation[2]+Request2=[3,0,1]
Need[2]=Need[2]-Request2=[0,2,1]
```

此时,各进程的资源分配状态如表8.2所示。

表8.2 各进程的资源分配情况

进程	Max			Allocation			Need			Available		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	7	5	3	0	1	0	7	4	3	2	3	1
P_2	3	2	2	3	0	1	0	2	1			
P_3	9	0	2	3	0	2	6	0	0			
P_4	2	2	2	2	1	1	0	1	1			
P_5	4	3	3	0	0	2	4	3	1			

(4) 下面执行安全检查算法,判断进程 P_2 的资源请求得到满足后系统是否处于安全状态。令

$$\text{Work}=\text{Available}=[2,3,1]$$
$$\text{Finish}[i]=\text{false}, i\in\{1,2,3,4,5\}$$

对于 Finish[2]=false,有 Need[2]≤Work,即[0,2,1]≤[2,3,1],所以 Work[j]=Work[j]+Allocation [2,j],$j\in\{1,2,3\}$;结果 Work=[5,3,2],Finish[2]=true。

对于 Finish[4]=false,有 Need[4]≤Work,即[0,1,1]≤[5,3,2],所以 Work[j]=Work[j]+Allocation [4,j],$j\in\{1,2,3\}$;结果 Work=[7,4,3],Finish[4]=true。

对于 Finish[5]=false,有 Need[5]≤Work,即[4,3,1]≤[7,4,3],所以 Work[j]=Work[j]+Allocation [5,j],$j\in\{1,2,3\}$;结果 Work=[7,4,5],Finish[5]=true。

对于 Finish[1]=false,有 Need[1]≤Work,即[7,4,3]≤[7,4,5],所以 Work[j]=Work[j]+Allocation [1,j],$j\in\{1,2,3\}$;结果 Work=[7,5,5],Finish[1]=true。

对于 Finish[3]=false,有 Need[3]≤Work,即[6,0,0]≤[7,5,5],所以 Work[j]=Work[j]+Allocation [3,j],$j\in\{1,2,3\}$;结果 Work=[10,5,7],Finish[3]=true。

显然,此时有 Finish[i]=true,$i\in\{1,2,3,4,5\}$,即系统仍处于安全状态,进程 P_2 的资源请求[1,0,1]可以得到满足,系统可以将 P_2 申请的资源立即分配给它。

8.5.5 死锁检测与恢复

如果在一个系统中,既没有采取死锁的预防措施,也没有采取死锁的避免措施,则系统中很有可能发生死锁。系统中一旦发生死锁,就要将其找到并将其解除。对于死锁的检测,需要解决两个问题,其一是死锁的检测方法,其二死锁的检测时机。

1. 资源分配图

进程的死锁问题可以用有向图更加准确直观地加以描述,这种有向图称作系统资源分配图(resource allocation graph),对资源分配图进行约简将得到重要的死锁定理。

定义 一个系统资源分配图是一个二元组 $G(V,E)$,其中 V 是节点集,E 是边集。节点集定义为 $V=P\cup R$,其中 $P=\{p_1,p_2,\cdots,p_n\}$为系统中所有进程所构成的集合,$R=\{r_1,r_2,\cdots,r_n\}$为系统中所有资源类所构成的集合。边集 $E=\{(p_i,r_j)\}\cup\{(r_j,p_i)\}$,其中 $p_i\in P$,$r_j\in R$。如果$(p_i,r_j)\in E$,则有一条由进程 p_i 到资源类 r_j 的有向弧,表示进程 p_i 申请资源类 r_j 中的一个资源实例。如果$(r_j,p_i)\in E$,则有一条由资源类 r_j 到进程 p_i 的有向弧,表示资源类 p_i 中的一个资源实例被进程 p_i 占有。形如(p_i,r_j)的边被称作申请边,形如(r_j,p_i)的边被称作分配边。

在图形中,将每一个进程表示为一个圆圈,每一个资源类表示为一个方框。由于一个资源类中可能含有多个资源实例,在方框中用圆点来表示同一资源类中的各个子资源实例。申请边只指向方框,表明申请时不指定哪一个资源实例;而占有边则由方框中的某一圆点引出,表明哪一个资源实例已被占用。

当进程 p_i 申请资源类 r_j 中的一个资源实例时,在资源分配图中增加一条申请边。当该申请可被满足时,该申请边立即被改为一条分配边。当进程释放该资源实例时,该分配边被去掉。

根据上述关于资源分配图的定义,容易证明:如果图中没有环路,则系统中没有死锁;

如果图中存在环路,则系统中可能存在死锁。

如果每个资源类中均只有唯一的一个资源实例,则环路的存在即意味着死锁的存在;如果存在一个由所有资源类所构成集合的子集合,该子集合中的每一个资源类均只有唯一的一个资源实例,则环路的存在即意味着死锁的存在。在上述情况下,环路是死锁的充分和必要条件。

如果每个资源类中包含有若干个资源实例,则环路并不一定意味着死锁的存在。此时,环路是死锁的必要但不是充分条件。

例 8.4 设进程集 P、资源类集 R 及边集 E 如下:

$$P = \{p_1, p_2, p_3\}$$

$$R = \{r_1(1), r_2(2), r_3(1), r_4(3)\}$$

$$E = \{(r_1, p_2), (r_2, p_2), (r_2, p_1), (r_3, p_3), (p_1, r_1), (p_2, r_3), (r_4, p_3)\}$$

其中,资源类 r_j 后括号中的数字表示资源实例的个数。对应的资源分配图如图 8.10 所示。

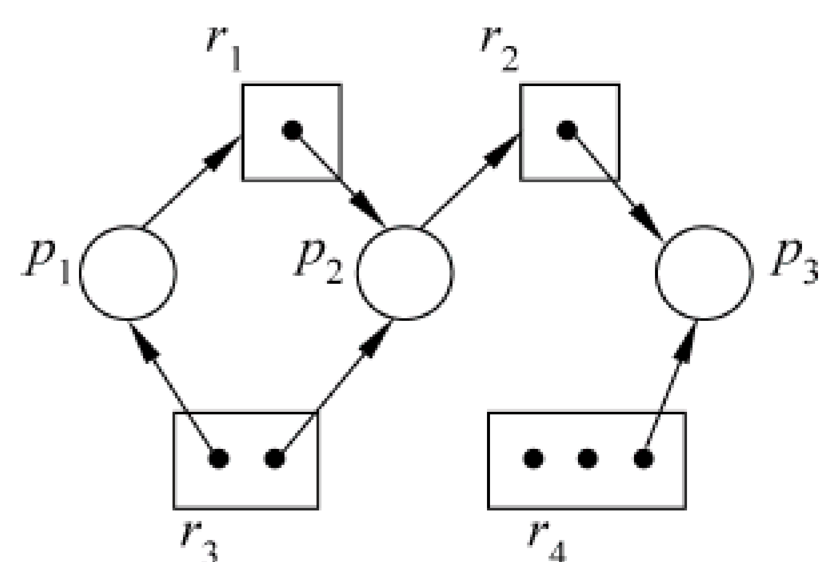


图 8.10 无环路的资源分配

此时进程 p_1 占有资源类 r_3 中的一个实例,等待资源类 r_1 中的一个实例;p_2 占有资源类 r_1 和资源类 r_3 中各一个实例,等待资源类 r_3 中的一个实例;p_3 占有资源类 r_2 和 r_4 中各一个实例。由于资源分配图中没有环路,因而不存在死锁。

例 8.5 对于图 8.10 所示的例子,假设进程 p_3 申请资源类 r_3 中的一个实例,由于没有空闲的资源实例,将增加一条申请边(p_3, r_3),形成图 8.11。此时,出现了两个环路:$p_1 \to r_1 \to p_2 \to r_2 \to p_3 \to r_3 \to p_1$ 和 $p_2 \to r_2 \to p_3 \to r_3 \to p_2$。进一步分析可以验证,此时系统已经发生死锁,且进程 p_1、p_2 和 p_3 都卷入了死锁。

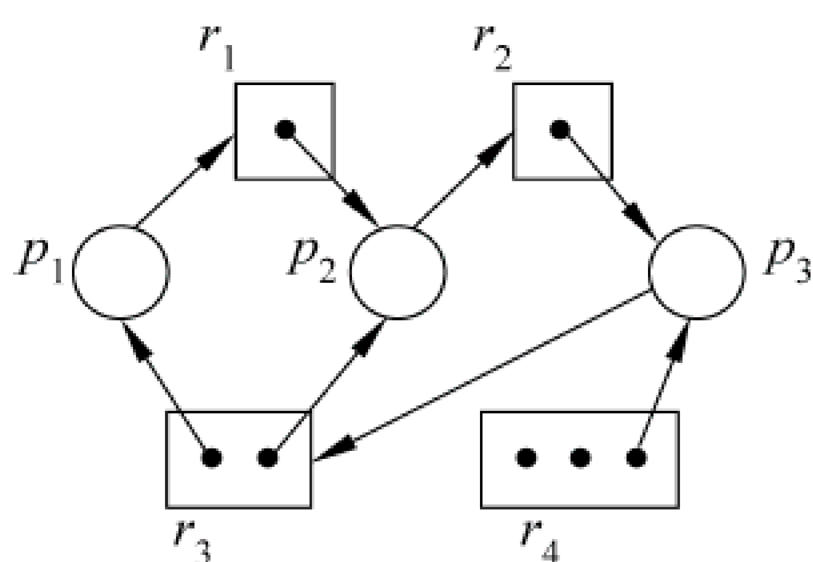


图 8.11 有环路的资源分配

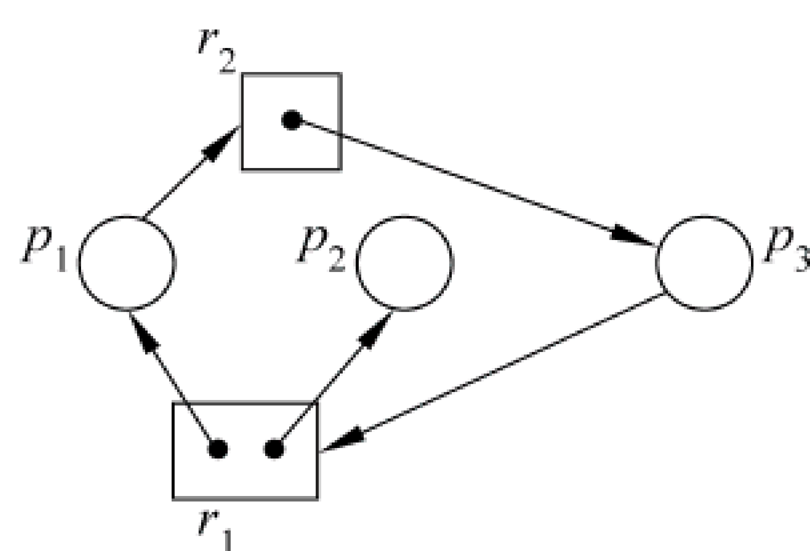


图 8.12 有环路但无死锁的资源分配

例 8.6 来看一下图 8.12 所示的例子。

图 8.12 中也有一个环路:

$$p_1 \to r_2 \to p_3 \to r_1 \to p_1$$

然而并不存在死锁。p_2 可能会释放资源类 r_1 中的一个资源实例,该资源实例可分配给进程 p_3,从而使环路断开。

综合上述分析可以看出,如果资源分配图中不存在环路,则系统中不存在死锁;反之,如果资源分配图中存在环路,则系统中可能存在死锁,也可能不存在死锁。这一结论对于处理死锁问题是很重要的。

2. 死锁定理

可以通过对资源分配图进行约简来判断系统是否处于死锁状态。资源分配图中的约简方法如下：

(1) 寻找一个非孤立且没有请求边的进程节点 p_i，若无则算法结束。

(2) 去除所有 p_i 的分配边使之成为一个孤立节点。

(3) 寻找所有请求边均可满足的进程 p_j，将 p_j 的请求边全部改为分配边。

(4) 转步骤(1)。

若算法结束时所有节点均为孤点，则称资源分配图为可以完全约简的，否则称之为不可完全约简的。

系统处于死锁状态的充分必要条件是资源分配图不可完全约简。这一结论称为死锁定理。

死锁定理 S 为死锁状态的充分必要条件是 S 的资源分配图不可完全约简。

例如，图 8.10 和图 8.12 所示的资源分配图是可以完全约简的，因而系统未死锁；图 8.11 所示的例子是不可完全约简的，因而系统已死锁。

3. 死锁检测算法

下面所介绍的死锁检测算法是由 Shoshani 和 Coffman 提出的，算法采用了如下数据结构。

Available：长度为 m 的向量，记录当前各个资源类中空闲资源实例的个数。

Allocation：$m \times n$ 的矩阵，记录每个进程当前占有各资源类中资源实例的个数。

Request：$m \times n$ 的矩阵，记录每个进程当前申请各资源类中资源实例的个数。如果 Request$[i,j]=k$，则进程 p_i 申请资源类 r_j 中 k 个资源实例。

两个向量间的关系和赋值操作的定义如银行家算法。为了表达简洁，将矩阵 Allocation 和 Request 的行看做是向量，并且分别标记为 Allocation$[i]$和 Request$[i]$。

死锁检测的算法如下：

(1) 令 Work 和 Finish 分别是长度为 m 和 n 的向量。初始时进行以下设置：

① Work=Available；

② 对于所有 $i=1,2,\cdots,n$，如果 Allocation$[i] \neq 0$，则 Finish$[i]$=false，否则 Finish$[i]$=true。

(2) 找满足下面条件的下标 i：

① Finish$[i]$=false；并且

② Request$[i] \leqslant$ Work。

如果不存在满足上面条件的 i，则转步骤(4)。

(3) Work=Work+Allocation$[i]$，Finish$[i]$=true。

转步骤(2)。

(4) 如果存在 i，$1 \leqslant i \leqslant n$，Finish$[i]$=false，则系统处于死锁状态，且进程 p_i 卷入了死锁。

步骤(1)的②对于当前不占有资源的进程直接将 Finish 置为 true，意味着检测略过了不占有资源的进程。

死锁检测的后继工作是恢复,而恢复与不占有资源的进程无关,因而死锁检测结果为下一步恢复提供了更加准确的信息。此外,只考虑占有资源的进程也缩小了检测范围,减小了系统开销。

当检测判断 Request[i]≤Work(步骤(2)的②时),进程 p_i 当前一定未被死锁。如果 p_i 以后不再申请新资源,则它可以进行完并且释放其所占有的资源;如果 p_i 以后还会申请资源,则它可能会被死锁,该死锁将在以后运行该算法时检测出来。

例 8.7 设系统中有 3 个资源类{A,B,C}。资源类 A 中有 7 个实例。资源类 B 中有 3 个实例,资源类 C 中有 6 个实例。又设系统中有 5 个进程{P_0,P_1,P_2,P_3,P_4}。假定某时刻系统中资源分配与申请情况如表 8.3 所示。

表 8.3 某时刻系统中资源分配与申请情况

进程	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
p_0	0	1	0	0	0	0	0	1	0
p_1	2	0	0	2	0	2			
p_2	3	0	3	0	0	0			
p_3	2	1	1	1	0	0			
p_4	0	0	2	0	0	2			

此时,系统不处于死锁状态,因为运行上述死锁检测算法可以得到一个进程序列<p_0,p_2,p_3,p_1,p_4>,它将使 Finish[i]=true,对于所有 1≤i≤n。

假定现在进程 p_2 发出请求(0,0,1),即申请资源类 C 中的一个资源实例,当前请求变化如表 8.4 所示。

表 8.4 p_2 发出请求后系统中进程请求资源的情况

进程	Request		
	A	B	C
p_0	0	0	0
p_1	2	0	2
p_2	0	0	1
p_3	1	0	0
p_4	0	0	2

此时,系统处于死锁状态,参与死锁的进程集合为{p_1,p_2,p_3, p_4}。

当系统中每个资源类都仅包含一个资源实例时,上述死锁检测算法可以得到简化。此时死锁的判断问题实质上就是一个环路检测问题。

4. 死锁检测时机

何时进行死锁检测主要取决于两个因素:

(1) 死锁发生的频率;

(2) 死锁所涉及的进程个数。

如果死锁发生的频率较高,则死锁检测的频率也应较高,否则影响系统资源的利用率,也可能使更多的进程被卷入死锁,对死锁进程所对应的事件也会带来影响。当然,死锁检测会增加系统的开销,影响系统效率。通常可在如下时刻进行死锁检测。

1）进程等待时检测

因为仅当进程发出资源申请命令且此申请不能立即满足时才有可能发生死锁，所以如果每当进程等待时便进行死锁检测，那么每当死锁形成时就能够被发现。当然，此时系统的开销将是很大的，与避免死锁的算法相近。

2）定时检测

为了减少死锁检测所带来的系统开销，可以采取每隔一段时间进行一次死锁检测的策略，如每隔一小时做一次死锁检测。此时，一次死锁检测可能会在资源分配状态图中发现多个死锁。

3）资源利用率降低时检测

为了减少盲目性，希望在系统可能已发生死锁的时刻进行死锁检测。大家知道，死锁的发生会使系统中可运行进程数量降低，因而会使处理机的利用率下降。所以，可在 CPU 的利用率降低到某一界限（如 45%）时开始进行死锁检测。

5．死锁的恢复

当死锁已经发生并且被检测到时，应当将其消除以使系统从死锁状态恢复过来。通常可采取如下策略消除死锁。

1）重新启动（system restart）

这是最简单、最常用的死锁解除方法。不过它的代价却是很大的，因为在此之前所有进程已经完成的计算工作都将付之东流，不仅包括参与死锁的全部进程，也包括并未参与死锁的全部进程。

2）终止进程（terminating processes）

通过终止参与死锁的进程并收回它们所占有的资源，死锁也能得以解除。这又有两种处理策略：

（1）一次性撤销所有参与死锁的进程。这种处理方法简单，但代价较高。例如，对于那些本身不占有任何资源的进程的撤销是不必要的。

（2）逐一撤销参与死锁的进程，即按照某种算法选择一个参与死锁的进程，将其撤销并收回其占有的全部资源，然后判断是否还存在死锁。如果是则选择并且淘汰下一个将被淘汰的进程，如此重复直至死锁解除。

3）剥夺资源（preempting resources）

即剥夺死锁进程所占有的全部或部分资源。在实现时又可分为两种情形：

（1）逐步剥夺。一次剥夺死锁进程所占有的一个或一组资源，如死锁仍未解除，则再继续剥夺，直至死锁解除为止。

（2）一次剥夺：一次性地剥夺参与死锁进程所占有的全部资源。

4）进程回退（process rollback）

所谓进程回退就是让参与死锁的进程回退到以前没有发生死锁的某个点处，并由此点开始继续，希望进程交叉执行时不再发生死锁。这似乎是死锁恢复的一个比较完善的方法，不过它所带来的开销是惊人的，因为要实现回退，必须“记住”以前某一点处的现场，而且该现场应当随进程的推进而动态变化，这需要花费大量的时间和空间。除此之外，一个回退的进程应当“挽回”它自回退点到死锁点之间所造成的影响，如修改某一文件，给其他进程发送消息，这些在实现时甚至是难以做到的。

例 8.8 设系统中有3种类型的资源(A、B和C)和5个进程P_1、P_2、P_3、P_4、P_5,A资源的数量为17,B资源的数量为5,C资源的数量为20。在T_0时刻系统状态如表8.5所示,系统采用银行家算法实施死锁避免策略。

表 8.5 在 T_0 时刻的系统状态

进　程	Max			Allocation		
	A	B	C	A	B	C
P_1	5	5	9	2	1	2
P_2	5	3	6	4	0	2
P_3	4	0	11	4	0	5
P_4	4	2	5	2	0	4
P_5	4	2	4	3	1	4
剩余资源数	A		B		C	
	2		3		3	

(1) T_0时刻是否为安全状态?若是,请给出安全序列。

(2) 若在T_0时刻进程P_2请求资源(0,3,4),是否能实施资源分配?为什么?

(3) 在(2)的基础上,若进程P_4请求资源(2,0,1),是否能实施资源分配?为什么?

(4) 在(3)的基础上,若进程P_1请求资源(0,2,0),是否能实施资源分配?为什么?

解:由题目所给的最大资源需求量和已分配资源数量,可以计算出T_0时刻各进程的需求源资数量Need,Need=Max−Allocation,如表8.6所示。

表 8.6 T_0 时刻的资源分配表

进程	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_1	2	1	2	3	4	7	2	3	3
P_2	4	0	2	1	3	4			
P_3	4	0	5	0	0	6			
P_4	2	0	4	2	2	1			
P_5	3	1	4	1	1	0			

(1) 利用银行家算法对T_0时刻的资源分配情况进行分析,可得此时的安全性分析情况如表8.7所示。

从T_0时刻的安全性分析中可以看出,存在一个安全序列{P_5,P_4,P_3,P_2,P_1},故T_0时刻的状态是安全的。

(2) 若在T_0时刻进程P_2请求资源(0,3,4),因为请求资源数(0,3,4)大于系统剩余资源数(2,3,3),所以不能分配。

表 8.7　T_0 时刻的安全性检测表

进程	Work			Need			Allocation			Work+Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P_5	2	3	3	1	1	0	3	1	4	5	4	7	true
P_4	5	4	7	2	2	1	2	0	4	7	4	11	true
P_3	7	4	11	0	0	6	4	0	5	11	4	16	true
P_2	11	4	16	1	3	4	4	0	2	15	4	18	true
P_1	15	4	18	3	4	7	2	1	2	17	5	20	true

(3) 在(2)的基础上,若进程 P_4 请求(2,0,1),按银行家算法进行检查:

P_4 请求资源(2,0,1)≤P_4 资源需求量(2,2,1)

P_4 请求资源(2,0,1)≤系统剩余资源数(2,3,3)

试分配并修改相应的数据结构,由此形成的资源分配情况如表 8.8 所示。

再利用安全性检测算法检查系统是否安全,可得到如表 8.9 所示的安全性检测表。

表 8.8　P_4 请求资源后的资源分配表

进程	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_1	2	1	2	3	4	7	0	3	2
P_2	4	0	2	1	3	4			
P_3	4	0	5	0	0	6			
P_4	4	0	5	0	2	0			
P_5	3	1	4	1	1	0			

表 8.9　P_4 请求资源后的安全性检测表

进程	Work			Need			Allocation			Work+Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P_4	0	3	2	0	2	0	4	0	5	4	3	7	true
P_5	4	3	7	1	1	0	3	1	4	7	4	11	true
P_3	7	4	11	0	0	6	4	0	5	11	4	16	true
P_2	11	4	16	1	3	4	4	0	2	15	4	18	true
P_1	15	4	18	3	4	7	2	1	2	17	5	20	true

从表 8.9 中可以看出,此时存在一个安全序列{P_4,P_5,P_3,P_2,P_1},故该状态是安全的,可以立即将 P_4 所申请的资源分配给它。

(4) 在(3)的基础上,若进程 P_1 请求(0,2,0),按银行家算法进行如下检查:

P_1 请求(0,2,0)≤P_1 资源需求量(3,4,7)

P_1 请求(0,2,0)≤系统剩余资源数(0,3,2)

试分配并修改相应的数据结构,由此形成的资源分配情况如表 8.10 所示。

表 8.10 P_1 请求资源后的资源分配表

进程	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_1	2	3	2	3	2	7	0	1	2
P_2	4	0	2	1	3	4			
P_3	4	0	5	0	0	6			
P_4	4	0	5	0	2	0			
P_5	3	1	4	1	1	0			

再利用安全性检测算法检查系统是否安全,系统可用资源数 Available 为(0,1,2),不能满足任何进程的资源需求,故系统进入不安全状态,此时系统不能将资源分配给 P_1。

8.6 Linux 中的线程同步

在 Linux 中,线程实际上就是进程,所以以前讨论的进程同步机制均可用于实现线程同步。Linux 中所提供的主要线程同步机制有自旋锁、信号量和条件变量,下面分别加以介绍。

1. 自旋锁

该方式通过对临界区加锁的方法保证某一时刻仅有一个线程在访问临界区,从而保证对共享资源的一致性访问。一般采用一个整数域作为锁。与通常锁状态的定义不同,当锁状态为 0 时,表示解锁状态,线程可以访问临界区资源;当锁状态为 1 时,表示处于上锁状态,说明此时有其他线程正在访问临界资源,线程应进入等待队列。系统循环检测锁的状态,直到变为 0,等待队列中优先级最高的线程就可以访问临界区了。自旋锁主要用于线程间细粒度的同步,即线程请求进入临界区的等待时间不宜过长。

2. 信号量

信号量用来解决因资源竞争而引起的数据不一致性问题。由于自旋锁值的反复循环测试,浪费许多 CPU 时间(这在 8.1.2 节中已作了详细说明),系统难以取得较好的性能,因此 Linux 利用信号量实现对临界区资源的访问,以克服上述缺点。信号量为一个整型变量,线程在信号量上施加的操作有两种:DOWN 操作和 UP 操作,分别实现信号量的减 1 和加 1 操作。在进入临界区之前,线程首先检查信号量的当前值,如果大于 0 则可以进行 DOWN 操作,并且该线程可以成功地进入临界区;否则线程进入阻塞状态,等待其他线程释放临界区后才能被唤醒。当线程结束对该资源的访问时,对信号量执行 UP 操作以释放资源。

3. 条件变量

与自旋锁和信号量不同,条件变量用来发送信号以表示某个操作已经完成,因此相对于等待资源锁定,它更适用于等待事件。而自旋锁和信号量主要用于控制对数据的访问。一

个线程因等待某个消息的到来或某个条件被设置而在某个指定的条件变量上被阻塞，该消息或标志是由另一个线程来发送或设置的，当消息到来或标志被设置后，该线程被唤醒。显然，对这个共享的消息或标志的访问需要一把自旋锁。因此，条件变量一般和一个额外的自旋锁配合使用。

8.7 Linux中的进程通信机制

Linux作为UNIX系统的克隆，它所提供的进程通信的方法和原理与UNIX一样，包括管道和信号等，并支持System V中所采用的IPC通信机制，即消息队列、信号量和共享存储器。其中消息队列主要用于进程间交换少量信息时的通信，共享内存用于进程间交换大量信息时的通信，信号量主要是结合共享内存以实现进程通信中的同步问题。上述通信机制用于位于同一主机内的不同进程间的信息交换，此外Linux还提供了套接字通信机制，它用于位于不同计算机上的进程间的通信。

8.7.1 管道

管道是Linux中最常见的通信机制。管道是单向的字节流，它实际上是在进程间开辟一个固定大小的缓冲区（Linux系统中为4KB），它将发送进程的标准输出和接收进程的标准输入连接起来，发送进程向管道的一端写数据，而接收进程则从管道的另一端读数据，即数据相当于从管道的一端流到另一端，这就是“管道”名字的由来。如图8.13所示，该种通信方式非常适合大数据量的信息交换。

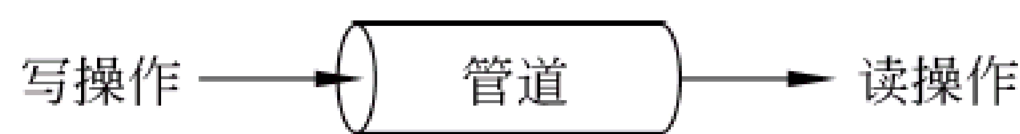


图8.13 管道操作示意图

在Linux中，管道的系统实现方式是由内核通过共享数据页来完成的。每个管道对应两个进程：读进程和写进程，分别完成对管道的读、写操作。在写进程对管道进行写操作的过程中，如果管道有足够的空闲空间且管道没有被读进程加锁，Linux首先对管道加锁，然后写进程将其地址空间中的数据复制到共享的数据页中；如果此时管道内没有足够的空间或管道已被读进程加锁，则写进程就进入等待队列，然后系统转内核的进程调度程序。此后，当有其他进程执行了V操作释放了所占用的管道空间或管道加锁解除后，写进程就被唤醒以完成对管道的写操作。写操作完成后执行V操作解锁管道。管道的读操作与管道的写操作相似，只是在管道的另一端读数据。读出的数据将从管道中移出，其他读进程不能再读到这些数据。对管道的I/O操作非常类似于对文件的I/O操作，进程实际上不知道它是正在对一个管道进行读/写，所以管道的读/写操作对用户来讲完全是透明的。

除了管道的系统实现方式外，用户也可以通过系统调用来完成对管道的操作，从而实现进程间的数据通信。用户可用系统调用pipe()创建管道，通过管道描述符调用write()和read()来实现对管道的写入或读出操作。

管道虽然为进程通信提供了一种有效方法，但是管道也存在一些不足。首先，因为读数据的同时也将数据从管道中移去，故管道不能同时对多个接收进程广播数据。其次，如果管道有多个读进程，那么写进程不能将数据发送给指定的读进程。同样，如果存在多个写进程，那么也无法判断数据是从哪个写进程发送来的。

8.7.2 System Ⅴ的 IPC 通信机制

Linux 把信号量、消息和共享内存定义为 System Ⅴ的 IPC 对象。各 IPC 对象由系统创建后返回一个唯一的 IPC 标识号，系统并以此标识号来唯一地标识所创建的 IPC 对象，这类似于用一个文件号来标识一个文件一样。这样，通过标识号使多个进程很容易共享 IPC 资源。各种 IPC 对象提供系统调用作为接口，进程只能通过系统调用来传递 IPC 标识以对相应的 IPC 对象进行访问。对象的创建者通过系统调用设置相应对象的访问权限。进程通信时必须先传递 IPC 标识，并由 ipcperms()函数（参见 ipc/util.c）确认权限后，才能访问通信资源。Linux 采用数据结构 ipc_perm 来表达 System Ⅴ的 IPC 对象，具体如下：

```
struct ipc_perm
{
    key_t key;                  /* 整型,为 0时表示 private,非 0时表示 public* /
    ushort uid;                 /* 资源拥有者的有效标识 * /
    ushort gid;                 /* 资源拥有者所在组的有效标识 * /
    ushort cuid;                /* 资源创建者的有效标识 * /
    ushort cgid;                /* 资源创建者所在组的有效标识 * /
    ushort mode;                /* 访问模式,其含义同文件访问模式 * /
    ushort seq;                 /* 序列号 * /
}
```

1. 消息队列

一个消息队列是一个由消息缓冲区所构成的链表，它允许一个或多个进程从中读出或写入消息。采用这种通信机制时，Linux 维护一个消息队列数组 msgque，每个数组元素为指向一个描述消息队列的 msqid_ds 结构指针，Linux 通过该结构管理消息队列。当创建新的消息队列时，系统将从内存中申请分配一个由 msqid_ds 结构描述的内存块，并将它插入到数组 msgque 中。该通信机制所用到的数据结构描述如下：

```
static struct msqid_ds msgque[MSGMNI];    /* ipc/msg.c* /
/* 每个消息队列占一个 msqid_ds 结构 * /
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg * msg_first;             /* 指向消息队列的第一条消息 * /
    struct msg * msg_last;              /* 指向消息队列的最后一条消息 * /
    time_t msg_stime;                   /* 最后发送时间 * /
    time_t msg_rtime;                   /* 最后接收时间 * /
    time_t msg_ctime;                   /* 最后修改时间 * /
    struct wait_queue * wwait;          /* 写消息进程的等待队列指针 * /
    struct wait_queue * rwait;          /* 读消息进程的等待队列指针 * /
    ushort msg_cbytes;                  /* 队列中消息的字节数 * /
    ushort msg_qnum;                    /* 队列中的消息数 * /
    ushort msg_qbytes;                  /* 队列中消息的最大字节数 * /
    ushort msg_lspid;                   /* 最后一个发送消息的进程的标识号 * /
    ushort msg_lrpid;                   /* 最后一个接收消息的进程的标识号 * /
```

```
}

/* 每条消息占一个 msg 结构 */
struct msg  {
    struct msg * msg_next;          /* 指向下一条消息的指针 */
    long msg_type;                  /* 消息类型 */
    char * msg_spot;                /* 消息文本的地址指针 */
    time_t msg_stime;               /* 发送此条消息的时间 */
    short msg_ts;                   /* 消息文本的长度 */
}
```

可见,Linux 中表示的结构中含有指下一个消息的指针 msg_next,每个消息队列均为一个单向链表。消息队列的组织形式如图 8.14 所示。

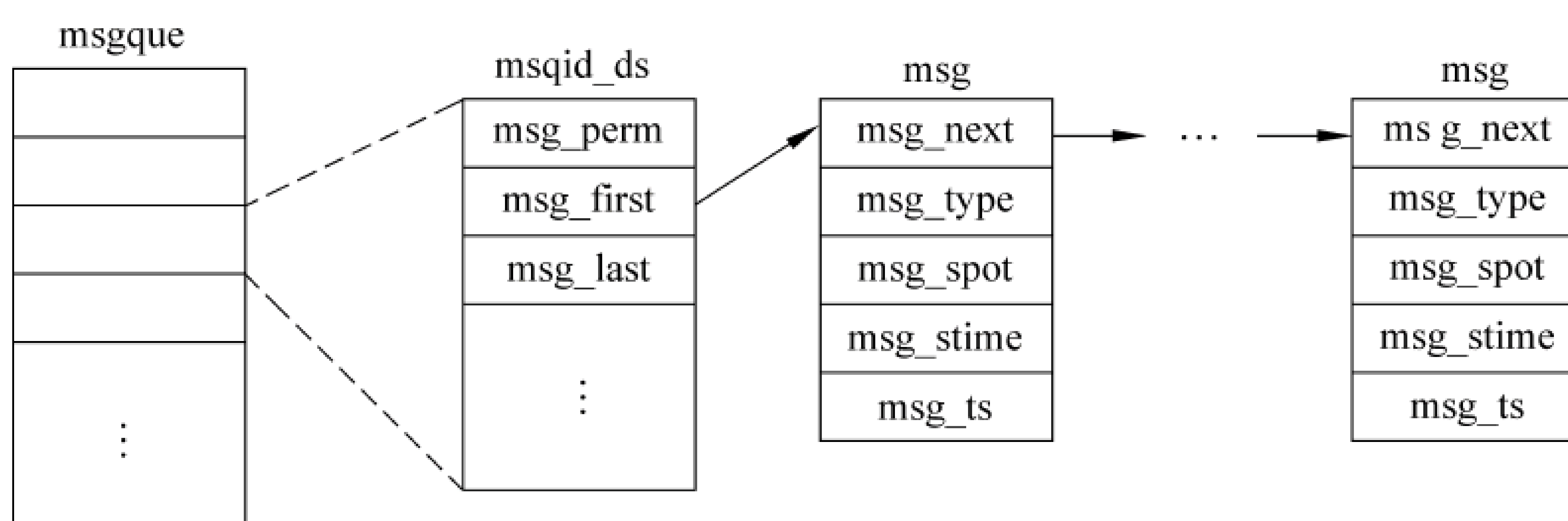


图 8.14 Linux 的消息等待队列

消息队列是进行读/写消息的存储空间,消息队列的创建以及读/写操作均是通过系统调用来实现的。下面介绍 Linux 中的几个相关的系统调用。

1) 系统调用 msgget()

使用 msgget()可以创建一个新的消息队列,或打开存取一个已经存在的消息队列。其原型为

```
int msgget(key_t key,int msgflg);
```

该系统调用若执行成功,则返回消息队列标识号;若执行失败,则返回−1。该系统调用原型的第一个参数为键值,用于表示消息的属性(公有或私有);第二个参数决定是创建还是打开一个消息队列,它的值为 IPC_CREAT 和 IPC_EXCL。

2) 系统调用 msgsnd()

该系统调用将一条消息传递给一个队列,其原型为

```
int msgsnd(int msqid ,struct msgbuf * msgp,int msgsz,int msgflg);
```

该系统调用原型的第一个参数 msqid 为消息队列标识号,由 msgget()返回。第二个参数 msgp 是一个指针,指向所要的消息缓冲区,具体定义形式如下:

```
struct msgbuf{
    long mtype;                     /* 消息类型,为一个整数 */
    char mtext[ ];                  /* 消息正文,类型可灵活定义,但长度要小于 8188B */
}
```

第三个参数 msgsz 表示消息的长度,以字为单位。第四个参数 msgflg 的取值有几种情况:当取值为 IPC_NOWAIT 时,表示如果消息队列已满,则不将消息写入队列,而直接将控制权返回给调用进程;当参数不指定为 IPC_NOWAIT 时,若消息队列中容纳不下这个新消息(如可能内存空间不足),则写消息的进程将暂时被添加到等待队列中,该等待队列由 msqid_ds 结构中的指针 wwait 所指定;直到这个消息队列中的消息被读走后,该进程才被唤醒。

3) 系统调用 msgrev()

该系统调用从消息传队列中的一条消息,其原型为:

```
int msgrev(int msqid ,struct msgbuf * msgp,int msgsz,long mtype,int msgflg);
```

该系统调用原型的第一个参数 msqid 为消息队列标识号,由 msgget()返回。第二个参数 msgp 是一个指针,用来指向用于存储所读到消息的缓冲区地址。第三个参数 msgsz 表示消息缓冲区的大小,以字为单位。

第四个参数 mtype 指定消息的类型,内核将搜索队列中相匹配的最早进入队列的消息,并且将该消息复制到由地址指针 msgp 所指定的缓冲区中;但当 msgp 为零时,即表示不考虑消息的类型,只返回最早进入队列的消息。

第五个参数 msgflg 取值为 IPC_NOWAIT 时,表示没有可取的消息,则直接将错误消息 ENOMSG 返回给调用进程;如果 msgflg 没有被指定为 IPC_NOWAIT 时,当无消息可取时,则该进程将被添加到读等待队列中,该等待队列由 msqid_ds 结构中的指针 rwait 所指定;当与 mtype 相符的消息进入队列时,该进程才能被唤醒。

消息通信机制允许一个或多个进程对消息队列进行读写操作。这种 IPC 机制通常使用在 C/S(Client/Server)模型中,多个客户向服务器发送请求消息,服务器读取消息并响应相应的请求。

消息队列和管道提供相似的服务,但消息队列的功能较强,并且解决了管道中所存在的一些问题。消息队列以一种不连续的方式传递数据,而不是采用无格式的字符流方式,因此,消息队列能够灵活地处理数据,特别是在传输小数据块的情况下效率更高。

2. 共享内存

这种机制允许不同的进程通过一块共享的内存区域来交换数据,从而达到相互通信的目的。该块共享内存由各个进程分别映射到各自的虚拟地址空间中,是各个进程虚拟地址空间的一个组成部分。该共享内存所对应的虚存页面出现在每个共享进程的页表中,但该页面在各进程的虚拟地址空间中可能处于不同的位置。映射后,这些共享内存就可以像常规内存一样被访问。因此,这种机制为进程通信提供了最快的实现方式,其通信效率比消息队列和管道等方式要高。

Linux 实现内存共享的方式有两种:系统实现方式和用户实现方式。当采用系统实现方式时,Linux 利用 shmid_ds 数据结构来表示每个新创建的共享内存区域,它描述共享内存的大小、有多少个进程在使用以及共享内存映射到其各自地址空间的方式等。这个数据结构被保存在 shm_segs 数组中。每个共享内存的进程必须通过系统调用将其连接到各自的虚拟内存上。但连接时虚拟内存并没有创建,而是在第一个进程试图访问它时才被创建的。当进程第一次访问共享虚拟内存时,发生页故障(即缺页中断)。这时 Linux 找到描述

该内存的 vm_area_struct 结构,其中包括了这类共享虚拟内存的处理例程的指针。共享内存的页故障处理代码在这个 shmid_ds 的页表项链表中查找,判断是否存在这个共享虚拟内存的页表项。如果不存在,就分配一个物理页,并为之创建一个页表项。此表项在填入该进程的页表中的同时也填入 shmid_ds 中,从而,当下一个进程试图访问这块内存时,共享内存页故障处理代码会让它使用同一物理页,这样,前后两个进程就可以通过同一物理页进行通信了。

当某一个进程不再共享虚拟内存时,它通过系统调用将自己的虚拟地址区域从链表中移去,并更新进程页表。当最后一个进程释放了自己的虚拟地址空间后,系统才能释放所分配的物理页。

用户进程若想采用共享内存方式与其他进程进行通信时,可以通过调用 shmget()来创建和获得一块共享内存区域;然后通过 shmat()把共享内存区域同其虚拟地址空间联系起来,用 shmdt()把共享内存区域和进程自己的地址空间分离出来,用 shmctl()对共享内存区域进行控制。

8.8 本章小结

现代操作系统均为多任务操作系统,系统内存在多个并发进程,这些并发进程可能同时访问同一共享资源,而系统中有些资源是仅允许互斥访问的,故提出临界资源和临界区的概念,以解决临界资源的互斥访问问题。同时系统内的多个并发进程可能相互合作来完成同一项任务,这些并发进程之间需要进行必要的信息交换,它们之间的执行具有一定的顺序关系,所以这些进程必须通过一定的同步措施确保它们协调地运行。为了解决上述问题,Dijkstra 等提出了 P、V 操作以及信号量等措施,用于解决进程互斥、同步和通信等问题,同时给出了生产者-消费者、读者-写者、哲学家进餐问题和理发师睡觉问题等几个典型的互斥、同步的解决方法。众所周知,系统资源是有限的,各并发进程可能因相互竞争资源而导致都进入阻塞状态,即出现死锁现象的发生。死锁是操作系统中所要考虑的一个相当重要的问题,死锁问题一般可通过预防、避免、检测与恢复等方法进行处理。Linux 中也采用信号量、管道、消息队列和共享存储器等进程通信方式,其中消息队列适合传输较小数据量时的通信,而共享内存则为进程通信提供了最快的实现方法。

习　题

1. 什么叫临界资源?什么叫临界区?
2. 详细解释并发进程间的两种制约关系。
3. 什么叫互斥?引入互斥的概念要解决什么问题?
4. 什么叫同步?引入同步的概念要解决什么问题?
5. 什么叫可剥夺资源和不可剥夺资源?
6. 打印机、磁带机、CPU 和存储器是可剥夺资源还是不可剥夺资源?
7. 理解信号量的概念,说明信号量各种取值的具体含义。
8. 用信号量和 P、V 操作描述对临界资源的互斥访问过程。

9. 用信号量和 P、V 操作描述计算进程和打印进程间的同步过程。

10. 为什么说进程的同步和互斥也是一种进程通信?

11. 在实现进程的互斥和同步过程中,各个 P 原语和 V 原语可以互换顺序吗?

12. 在一个仅允许单向行驶的单排车道十字路口,只允许车辆从南向北和从西向东行驶,如图 8.15 所示。为了安全起见,每次仅允许一辆汽车通过十字路口。当有车辆通过路口时其他车辆必须等待,当无车辆在路口行驶时则一次仅允许一辆汽车通过。请用 P、V 原语及信号量实现十字路口的交通控制系统,并描述具体的控制算法。

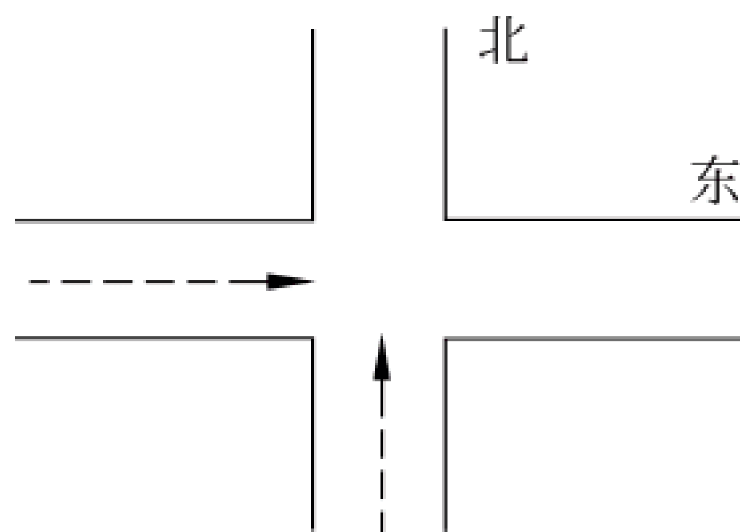


图 8.15 允许单向行驶的单排车道十字路口

13. 设存在 3 个过程 get、copy 和 put 分别对缓冲区 S 和 T 进行操作,其中 get 负责将数据块存入缓冲区 S,copy 负责从缓冲区 S 读出数据并复制到缓冲区 T 中,put 负责从缓冲区 T 中读出数据并打印,如图 8.16 所示。请用 P、V 操作描述上述 3 个过程。



图 8.16 输入-缓存-输出问题

14. 烟-吸烟者问题。考虑一个系统有 3 个吸烟者进程和 1 个代理进程。每个吸烟者持续不停地卷烟和吸烟。但是为了卷烟和吸烟,吸烟者必须同时拥有 3 种材料:烟丝、卷烟纸和火柴;3 个吸烟者分别拥有其中的一种。代理可以提供足够的 3 种材料,但每次仅将两种不同的材料放在桌子上供吸烟者使用。手中拥有一种材料的吸烟者要卷烟和吸烟时就向代理发信号,然后代理将所需要的两种材料放在桌子上;如此循环重复这个过程。请写出代理和吸烟者间的同步算法。

15. 有一位材料保管员负责管理笔和纸。另有 A、B 两组学生,A 组学生每人手中都备有纸,而 B 组学生每人手中都备有笔,任一学生只要能得到其他一种材料就可以写信。有一个可以放一张纸或一支笔的小盒,当小盒为空时,保管员就可以任意放一张纸或一只笔供学生使用。当小盒中有物时,每次仅允许一个学生从中取出。假设管理员手中所拥有的笔和纸的数量等于 A、B 两组学生各自所需要的总数,请用信号量描述上述过程。

16. A、B 两人共同使用一个报箱,该报箱每次仅能容纳一份报纸。A 订阅《生活报》,B 订阅《晚报》,分别由投递员 C 和 D 投递。请用 P、V 操作描述他们的同步程序。

17. 怎样理解死锁现象?引起死锁的必要条件是什么?

18. 引起系统死锁的两个根本原因是什么?

19. 死锁的解除方法有哪些?

20. 什么叫系统的安全状态?一般采用什么方法保持系统处于安全状态?

21. 有 6 位学者均匀地坐在圆桌旁,有 3 张纸和 3 只笔分别相间地摆放在相邻两位学者中间的桌面上,如图 8.17 所示。当一个学者想要记录时,他需要同时拥有他两侧的笔和纸时才能进行;如果此时他一侧的笔或纸正被他的邻座所使用,则他只能等待;学者用完纸或笔后,必须放回原位置。请用 P、V 操作描述上述过程的同步算法。

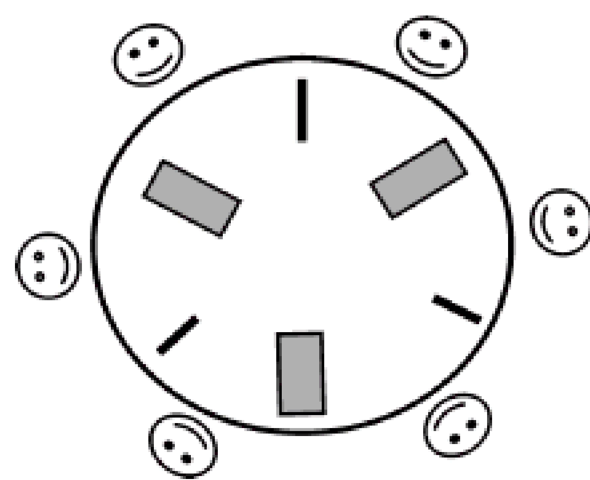


图 8.17 学者记录问题

22. 公交车上司机负责驾驶汽车，而售票员负责开关车门，他们之间必须协同工作。一方面售票员关好车门并通知司机后，司机才能开车；另一方面，司机将车停稳并通知售票员后，售票员才能打开车门上下乘客。假设某辆公交车上有一名司机和两名售票员，每个售票员各负责一个车门，请设适当的信号量，并用P、V原语描述他们的同步过程。

23. 设系统中有 n 个并发进程，共同竞争共享资源X，且每个进程都需要 m 个X类资源，为了保证系统不会发生死锁，资源X至少需要多少个？

24. 一台计算机配有6台磁带机，它们由 n 个进程竞争使用，每个进程最多需要2台磁带机，请问 n 为多少时系统不会发生死锁？请解释原因。

25. 一个系统中存在2个进程和3个同类资源X，每个进程最多需要2个X类资源，请判断是否会发生死锁？请解释原因。

26. 仅涉及单一进程的死锁会发生吗？请解释原因。

27. 一个系统中存在5个进程和4种共享资源，资源的当前分配和需求情况如表8.11所示。

(1) 试问该状态是否安全？

(2) 若进程 P_2 提出资源请求(1,2,2,0)后，系统能否将资源分配给它？

表8.11 各进程的资源分配情况

进程	Allocation				Need				Available			
	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4
P_1	0	0	3	2	0	0	1	2	1	6	2	2
P_2	1	0	0	0	1	7	5	0				
P_3	1	3	5	4	2	3	5	6				
P_4	0	0	3	2	0	6	5	2				
P_5	0	1	1	4	0	6	5	6				

28. 化简如图8.18所示的资源分配图，并判断是否为死锁状态。

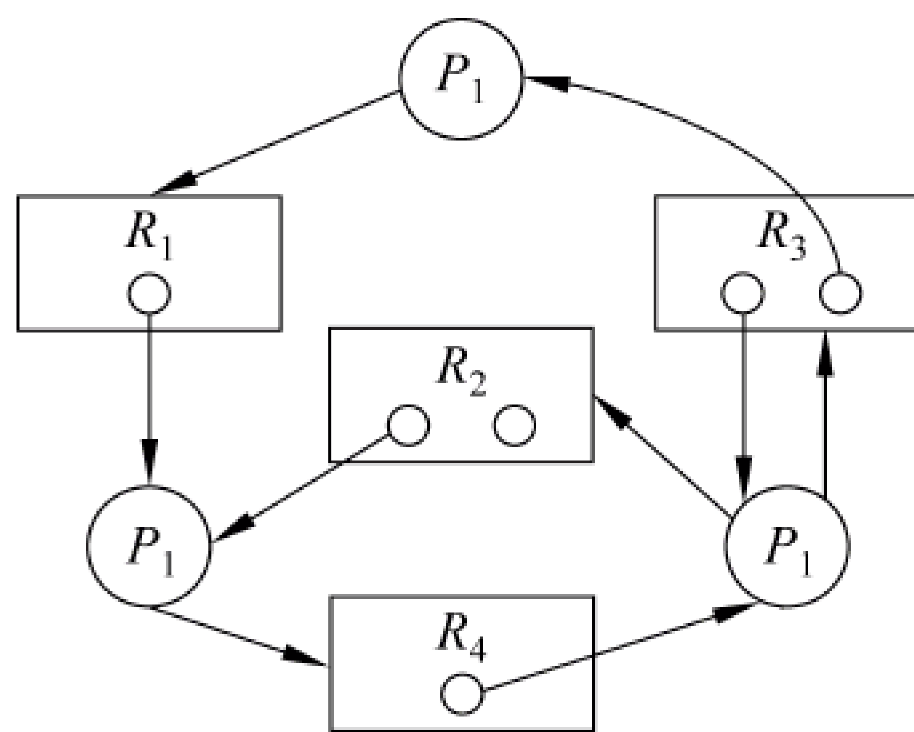


图8.18 资源分配图

29. 解释消息传递通信的两种实现方式。

30. 说明高级通信和低级通信间的区别。

31. 说明Linux中的管道通信机制及其优缺点。

32. 解释Linux中的消息传递通信机制。

33. 解释Linux中的共享内存通信机制。

第 9 章　其他几种操作系统简介

9.1　安全与安全操作系统

9.1.1　安全

1. 影响安全的 3 个最重要的方面

影响安全的因素有许多方面，最重要的 3 个方面是威胁、入侵和意外数据丢失。

1）威胁

从安全方面看，计算机系统有 3 个目标，与之对应的有 3 种威胁。

第一个目标是数据保密性，即保密数据得到保密。其对应的威胁是数据泄密。

第二个目标是数据的完整性，非授权用户在没有拥有者的授权情况下不能修改数据。其对应的威胁是数据的篡改。

第三个目标是系统有效性，没有人能够干扰系统，使系统不能运行。目前像拒绝服务的攻击越来越普遍。

安全中的另一个问题是隐私，即防止个人信息被他人使用。该问题涉及法律和道德问题，在此不进行说明。

2）入侵者

就安全方面来说，把进入与本人无关地方的人称为入侵者，有时也称为敌人。入侵者的行为有两种：

（1）被动入侵者想读没有授权的文件。

（2）积极入侵者具有更多的蓄意性，他们想改变未经授权的数据。

设计一个能抵御入侵者的系统，重要的是了解入侵者。常见的 4 种入侵者如下：

（1）偶然窥探的非技术人员；

（2）窥探的系统内人员；

（3）企图赚钱的人；

（4）商务和军事间谍。

3）意外数据丢失

除了蓄意入侵者造成的数据丢失之外，还有意外造成的数据丢失。引起数据丢失的常见意外情况有以下几类：

（1）天灾：火灾、洪水、地震、骚扰、战争或磁盘和磁带的损坏。

（2）硬件或软件错误：CPU 故障、不能读磁盘或磁带以及程序的 bug 等。

（3）人为错误：不正确的数据输入、安装错误的磁盘或磁带、错误的程序运行以及磁盘或磁带的丢失等。

防止这些事件发生的有效方法是适时备份数据。

2. 保证安全的基本措施

保证安全的基本措施有密码术和用户认证。

1）密码术

数据加密通常被认为是为数据存储和传输提供保密性的最佳方法。加密是使用一种算法把数据从一种形式(明文)变换到另一种形式(密文),在变换过程中要使用一个或多个加密密钥。如果不使用正确的密钥来解密数据,那么存储或传输的加密过的结果数据是没有意义的。要使数据在一个不可信网络上保持机密,就必须对数据进行加密。

通常在两种情况下需要对数据进行加密:当需要安全的存储以及传输秘密的或者需要高度机密性的信息时,需要使用加密。

加密方法主要有两种:私有密钥和公开密钥。数字签名是把私有密钥和公开密钥两种加密方法结合在一起的一个应用实例。数字签名主要用在电子邮件、文档提交、公共网络上的电子商务以及电子数据交换等领域。

2）用户认证

用户认证就是验证用户的身份,主要的验证方法如下:

(1）使用口令认证。使用最广的一种认证方式是要求用户输入登录名和口令。

(2）使用物理对象的认证。该方法是检查只有用户知道并且是用户有的某些物理对象。传统的已经用了几个世纪的门钥匙就是物理对象认证的一种方式。现在使用的物理对象是塑料制卡。

(3）使用生物技术的认证。该方法是测量用户很难被伪造的生物特征。例如,在终端上的指纹或声纹阅读设备可以鉴别用户的身份。

(4）计数测量。该方法是统计登录者连续登录的次数,在登录若干次之后没有成功,则拒绝登录。

3. 对系统的进攻

对系统的攻击是造成系统不安全的主要因素,对系统的攻击主要有来自系统内的和来自系统外的。

1）来自系统内的攻击

来自系统内的攻击主要指攻击者直接操纵系统的终端而进行的攻击。下面是几种来自系统内的攻击的方式:

(1）特洛伊木马;

(2）登录欺骗;

(3）逻辑炸弹;

(4）天窗;

(5）缓冲区溢出;

(6）普通的安全攻击;

(7）安全漏洞。

2）来自系统外的攻击

对于连接到 Internet 或其他网络上的计算机来说,攻击可以来自网络上的远距离的一台计算机,这种攻击称为外部攻击。来自系统外的攻击主要有以下几类:

(1）病毒

计算机病毒是一种传染其他程序的程序,它通过修改其他程序使之包含病毒自身精确副本或者可能的演化版本变形或者其他繁衍体。最重要的病毒按其存在方式类型可分为以

下五类：

① 寄生病毒；

② 常住内存病毒；

③ 引导区病毒；

④ 隐形病毒；

⑤ 多形态病毒。

其中，有以下两种特殊的病毒：

① 宏病毒。这是一类感染范围广泛且品种繁多的病毒。

② 电子邮件病毒。这是近年来随着网络使用者的急剧扩展而迅速蔓延的一种病毒。

(2) Internet 的蠕虫

在1988年11月2日，上千台连接到 Internet 上的 VAX 和 UNIX 系统开始神秘地瘫痪。它们感染了一种称为蠕虫的计算机病毒。该蠕虫启动一个又一个的进程，从而迅速耗尽受感染的计算机资源，使受感染的计算机瘫痪。

(3) 动态代码

现在许多网页包含被称为 applets 的一段小程序。当包含有 applets 的网页被下载时，applets 被获得并执行。当进程输入一个 applets 或其他动态代码到它的地址空间里并执行它，applets 作为一个致命的用户进程的一部分运行，拥有用户的所有权限，包括读、写、删除、加密磁盘上的文件和发送电子邮件等。

(4) Java 的安全

Java 程序设计语言和伴随的运行系统被设计为允许一次性编写和编译程序，然后以二进制的形式在 Internet 上传输，可以在任何支持 Java 的计算机上运行。由于 Java 编译后的程序在 Internet 上传播，因而 Java 的安全直接影响到 Internet 的安全，故从开始安全就作为 Java 的一部分。

4. 保护机制

系统中存在许多潜在的问题，有些是技术问题，有些是非技术问题。这里讨论一些保护文件和其他对象的详细的技术方法，这些技术使得原则(哪些数据应该进行哪些方面的保护)和机制(系统怎样实现这些原则)更加明确地得以区分。常用的保护机制有以下3种。

1) 保护域

一个操作系统包含许多需要保护的对象。这些对象可以是硬件，也可以是软件。每一个对象有一个唯一的名字和一个有限的操作系列，如 read 和 write 操作适合文件，P 和 V 操作适合同步信号。

一个域是(对象，权限)对。每一对说明一个对象和在其上可进行的操作。

在图9.1中显示了3个域以及域中的对象和每一对象的权限(R—Read，W—Write，X—Execute)。

在每一固定时刻，每一进程在某一保护域中运行。换句话说，进程可能访问某些对象，对某一对象而言它具有某些权限。在执行过程中对象可以从一个域切换到另一个域中。记录域的一种常见的方法是存取控制矩阵。对应图9.1的存取控制矩阵如图9.2所示。

利用如图9.2所示的矩阵存储域太浪费空间。两种实际的存储方法是仅存储非空元素：一种是按列存储矩阵，另一种是按行存储矩阵。

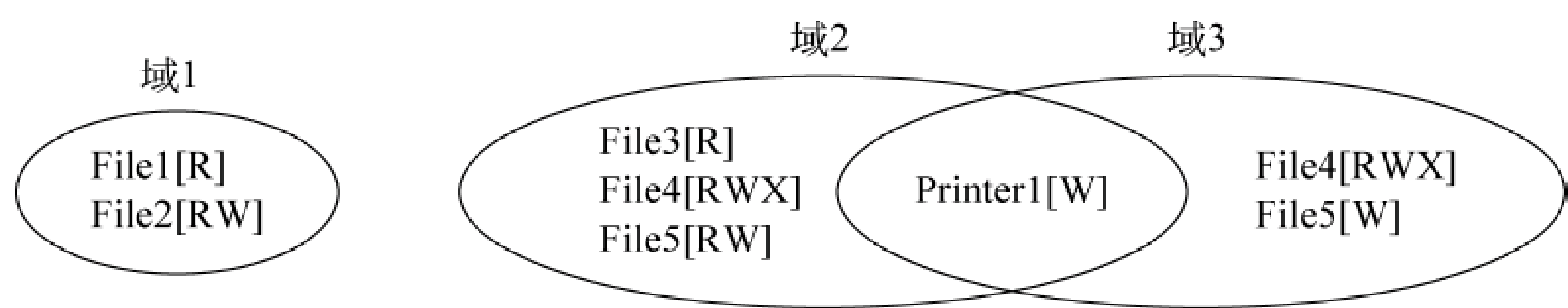


图 9.1　域及域中对象的权限

域＼对象	File1	File2	File3	File4	File5	File6	Printer1	Printer2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

图 9.2　管理文件访问的存取控制矩阵

2）访问控制列表

访问控制列表是按列存储矩阵的方法，由结合每一个对象的列表组成，该列表包含所有的域。这种列表称为访问控制列表(Access Control List，ACL)。图 9.3 是一个存取控制矩阵的示例，它所对应的访问控制列表如图 9.4 所示。

域＼对象	F1	F2	F3
1	RW	R	
2	R	RW	RWX
3		R	RX

图 9.3　存取控制矩阵

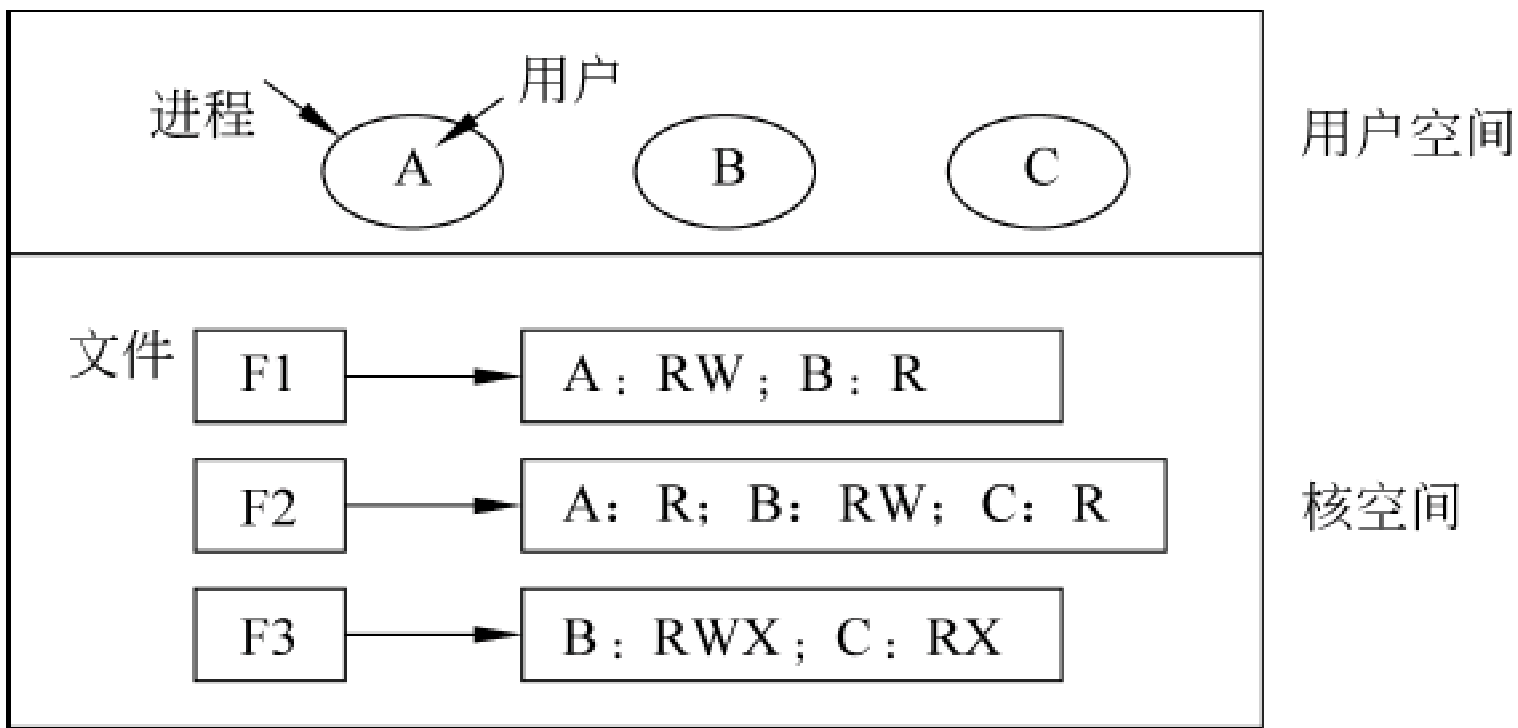


图 9.4　管理文件访问的访问控制列表

这里设 3 个进程 A、B 和 C，每个进程分别属于域 1、2 和 3，3 个文件 F1、F2 和 F3。为了简单起见，假设每个进程只响应一个用户，这样有 3 个用户 A、B 和 C，用安全术语来说用户被称为主体(subject)，它所拥有的事务(如文件)称为客体(object)。每一个客体有一个

与它相关的ACL。

3) 权能列表

对存取控制的另一种变换方式是按行进行。利用这种方法时,同每一进程相关的是可以被访问的对象列表,同时说明在该对象上可以进行的操作。换句话说,就是以用户为单位组织的"域"表。该表称为权能(capality)列表,该表当中的每一条目称为权能。对应图9.3所示矩阵的权能列表如图9.5所示。

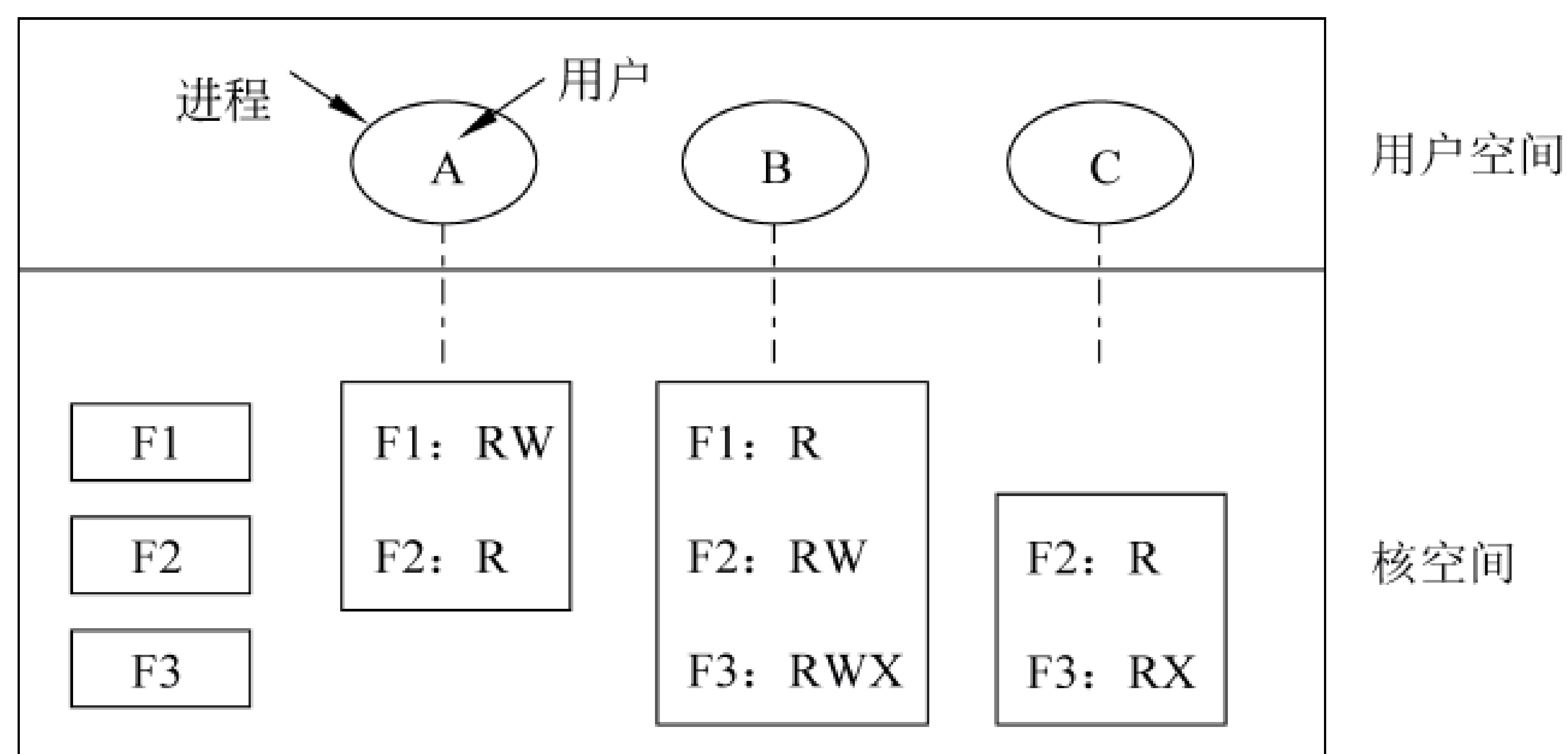


图9.5 管理文件访问的权能列表

可以采取下列3种方法防止用户对权能列表的修改:

(1) 要求一个标记结构。它是一种硬件设计,在结构中内存的每一个字有一个额外的标记位用来说明该字是否包含权能。

(2) 把权能列表保存在操作系统内。

(3) 把权能列表保存在用户空间中,但以加密的方式管理权能,所以用户不能修改它。

权能列表能提供一种面向对象的保护模式。基于权能的系统具有基于ACL的系统所无法支持的安全特性:

(1) 最小特权。进程不拥有执行任务所需之外的权能。在ACL系统中,进程根据用户的身份获得权限,因而由某一用户发起的所有进程拥有相同的权限。

(2) 选择性授权访问。父进程能够选择其所拥有的权限传给子进程。

(3) 权限传递控制。进程只能经授权的受控通道获得额外的权限。

5. Linux的安全问题

在Linux系统中采取了一系列的安全措施,从而使它的安全性较高。但Linux系统还是有许多安全漏洞,因而具有安全隐患。

1) Linux的安全措施

在Linux系统中为了保证系统的安全主要采取了7项安全措施。

(1) 标识

对于登录到系统中的每一用户,都有一个用户标识符来标识其身份,该标识符是系统管理员创建用户账户时确定的,并在内部分配一个唯一的标识号。

Linux系统中的用户分为超级用户(只有一个)和一般用户。作为超级用户,可以控制一切:用户账号、文件和目录、网络资源。一般用户只可进行权限范围内的操作。

(2) 鉴别

用户名是一个标识,它告诉系统用户是谁。口令才是确认的证据。用户登录时,需输入

口令来让系统鉴别其身份。在创建用户时,系统要求用户给定一个口令,如给定的口令不符合要求或系统判别其容易被破译,则要求用户重新给定一个。

(3) 存取控制

文件和目录的存取权限分为读、写和执行。对于文件和目录,不同的用户有不同的访问权限。

(4) 审计

Linux 系统的审计机制监控系统发生的事件,以保证安全机制正确工作并及时对系统异常报警提示。审计结果通常写在系统的日志文件中。大部分日志信息存放在/var/log 目录中。相应地 Linux 有许多日志工具,如 lastlog 跟踪用户登录,last 报告用户的最后登录。Linux 系统的审计达到 C2 级。

(5) 密码

当前 Linux 系统中常用的加密程序有 crypt、des 和 pgp。Linux 可以提供一些点对点的加密方法,以保证传输中的数据。一般情况下,当数据在 Internet 中传输时,可能要经过许多网关。在这一过程中,数据很容易被窃取。各种添加的 Linux 应用程序可以进行数据加密和打乱数据的操作,这样即使数据被截获,也是没有用的一堆乱码。

(6) 网络监视和入侵检测

入侵检测技术是一项较新的技术。标准的 Linux 发布版本也是最近才配备了这种工具。利用 Linux 配备的工具和从 Internet 下载的工具,可以使系统具备较强的入侵检测能力。例如,利用嗅探器可以有效地监听网络上的信息,利用扫描仪可以检测安全漏洞等。

(7) 备份和恢复

无论采取怎样的安全措施,都不能消除系统崩溃的可能性。系统的安全性和可靠性是与备份有关的。定期备份是一项非常重要的事情,它使用户在系统崩溃之后将损失减到最小。

Linux 系统中的备份程序有 dump、restore 和 backup 等。

2) Linux 的安全漏洞

任何一个操作系统都存在安全漏洞,Linux 也不例外。Linux 的安全漏洞包括不可读文件被跟踪、adduser/useradd 设计错误以及 RedHat Linux ping 缓冲区溢出漏洞等等。

9.1.2 安全操作系统

操作系统的安全性能直接影响到建立在此基础上的各种软件系统的安全性,因此一个安全的操作系统是建立在其上的软件系统的根基。

1. 安全操作系统的概念及重要性

根据计算机软件系统的组成,软件安全可划分为应用软件安全、数据库安全、网络软件安全和操作系统安全。操作系统用于管理计算机资源,控制整个系统的运行,它直接和硬件打交道,并为用户提供接口,是计算机软件的基础。数据库通常是建立在操作系统之上的,若没有操作系统安全机制的支持,数据库就不可能具有存取控制的安全可信性。在网络环境中,网络的安全可信性依赖于各主机系统的安全可信性,而主机系统的安全性又依赖于其操作系统的安全性。因此,若没有操作系统的安全性,就没有主机系统的安全性,从而就不可能有网络系统的安全性。计算机应用软件都建立在操作系统之上,它们都是通过操作系

统完成对系统中信息的存取和处理。因此，操作系统的安全是整个计算机系统安全的基础，没有操作系统安全，就不可能真正解决数据库安全、网络安全和其他应用软件的安全问题。

安全操作系统（Secure Operating System，SOS）是对所管理的数据和资源提供适当的保护级别，有效地控制硬件和软件功能的操作系统。通常一种安全操作系统从开始设计时就充分考虑到系统的安全性；另一种是基于一个通用的操作系统，专门进行安全性改进或增强，并通过相应的安全性评测。

2. 安全操作系统的发展

按技术成果的特点把安全操作系统的发展过程划分为 4 个阶段：奠基阶段、食谱阶段、多政策阶段和动态政策时期。

1）奠基阶段

该阶段始于 1967 年 Adept-50 项目的启动之时，在这一时期，安全操作系统经历了从无到有的探索过程，安全操作系统的基本思想、理论、技术和方法逐步建立。这一阶段开始的主要标志是美国国防部计算机安全特别部队组建，Adept-50 项目启动。发展的促进因素是美国军方意识到资源共享的计算机系统的安全威胁。主要特点是摸索基本的思想、技术和方法。主导开发方法是基于威胁的方法。这一阶段主要的代表思想是 Adept-50 主体、客体、访问控制矩阵、引用监控机、安全核、隐蔽通道 BLP 模型、权能、ACL、系统设计原则和操作系统保护理论。主要的代表系统有 Multics、Mitre 安全核、UCLA 安全 UNIX、KSOS 和 PSOS。技术难点是创建基本的思想、技术和方法。主要问题是初始创建难度大。

2）食谱阶段

该阶段始于 1983 年美国的 TCSEC 标准（简称橘皮书）颁布，其特点是人们以 TCSEC 为蓝本研制安全操作系统，有 C2～A1 各等级系统推出。发展的促进因素是美国国防部要求被采纳的系统必须符合 TCSEC 标准。主导开发方法是基于标准的方法。这一阶段的代表思想是可信计算基、系统安全等级。主要的代表系统有 LINUSIV、安全 Xenix、System/V MLS、TUNIS 和 ASOS。技术难点是高安全等级系统研制。主要问题是系统与变化的应用要求脱节。

3）多政策阶段

该阶段始于 1993 年，这一阶段的特点是人们超越 TCSEC 标准的范围，在安全操作系统中实现多种安全政策，系统研制有抽象框架，无蓝本。这一阶段开始的主要标志是美国国防部制定并推出了新的安全体系结构 DGSA 框架，它的显著特点之一是对多级安全政策支持的要求，这为安全操作系统的研究提出了新的挑战，促进安全操作系统研究进入了新阶段。发展的促进因素是网络应用范围扩大，美国国防部要求被采纳的系统必须与 DGSA 一致。主导开发方法是基于威胁的方法。这一阶段主要的代表思想是多安全政策共存、算法的安全政策支持结构、访问控制程序和看守员；这一时期主要的代表系统有安全服务器、客体管理器和 DTOS。技术难点是多种安全政策的共存。主要问题是确定系统支持结构有难度。

4）动态政策时期

该阶段始于 1999 年，这一时期的特点是使安全操作系统支持多种安全政策的动态变化，实现安全政策的多样性。这一阶段开始的主要标志是美国国家安全局等推出 Flaks 安全体系结构。发展的促进因素是真实世界的环境要求系统应该处理政策的变化特征。主导

开发方法是基于威胁的方法。这一时期主要的代表思想或代表系统是安全政策灵活性和安全政策配置。这一时期主要的代表系统有 Flask 和 SELinux。技术难点是政策变化特点的处理。

3. 安全操作系统的一般模型

说一个操作系统是安全的，是指它满足某一给定的安全策略(security policy)。进行安全操作系统的设计和开发时，也要围绕一个给定的安全策略进行。安全策略是指有关管理、保护和发布敏感信息的法律、规定和实施细则。例如，可以将安全策略定义为：系统中的用户和信息被划分为不同的级别，一些级别比另一些级别高。

安全模型就是对安全策略所表达的安全需求的简单、抽象和无歧义的描述，它为安全策略与其实现机制的关联提供了一种框架。安全模型描述了对某个安全策略需要用哪种机制来满足，而模型的实现则描述如何把特定的机制应用于系统中，从而实现某一特别的安全策略所需的安全保护。

1) 安全模型的特点及分类

安全模型有以下特点：

(1) 精确的、无歧义的；

(2) 简单的、抽象的，容易理解；

(3) 一般性的，只涉及安全性质，不过多牵扯系统的功能或其实现细节；

(4) 安全策略的明显表现。

安全模型一般分为两种：形式化的安全模型和非形式化的安全模型。非形式化安全模型仅模拟系统的安全功能，形式化安全模型则使用数学模型精确地描述系统的安全功能。

2) 安全模型介绍

目前被公认的安全模型主要有以下几种：

(1) 状态机模型(state machine model)

用状态机语言将安全系统描绘成抽象的状态机，用状态变量表示系统的状态，用转换规则描述变量变化的过程。状态机模型早就用于描述其他系统，但用于描述通用操作系统的所有状态变量几乎是不可能的。状态机安全模型通常只能描述安全操作系统中若干与安全相关的主要状态变量。著名的 BLP 模型是一个状态机模型。

(2) 信息流模型(flow model)

信息流模型用于描述系统中客体间信息传输的安全需求，根据客体的安全属性决定主体对它的存取操作是否可行。信息流模型不是检查主体对客体的存取，而是试图控制从一个客体到另一个客体的信息传输过程。许多人重视信息流模型，只是因为它可用于寻找隐蔽通道。信息流模型中的一个典型代表是 D. Denning 提出的信息流的格模型。

(3) 无干扰模型(noninterference model)

无干扰模型将系统的安全需求描述成一系列主体间操作互不影响的断言，要求在不同存储域中，操作的主体能够防止由于违反系统的安全性质导致的相互间的影响，如要求高安全级的操作不干扰(影响)低安全级主体的活动。它曾用于 Honeywell 公司的安全 Ada 研究项目。

(4) 不可推断模型(nondeducibility model)

该模型提出了不可推断性的概念，要求低安全级用户不能推断出高安全级用户的行为。

（5）完整性模型（integrity model）

目前公认的两个完整性模型是：

① Biba模型。它通过完整级（integrity level）的概念，控制主体“写”访问操作的客体范围。

② Clark-Wilson模型。它针对完整性问题，对系统进行功能分隔和管理。

实际上，在操作系统安全中常涉及的安全模型主要有状态机模型和信息流模型。

4. 安全操作系统的开发方法

安全操作系统的开发方法有两种：一种方法是采用从头开始建立一个完整的安全操作系统，第二种方法是基于非安全操作系统（ISOS）开发安全操作系统的方法。

前一种方法用得非常少，这里不做介绍。

第二种开发方式是目前普遍采用的一种方式，它可以把现有的操作系统合理地利用起来。利用这种方式开发安全操作系统一般有3种方法：虚拟机法、改进/增强法和仿真法。

1）虚拟机法

在现有操作系统和硬件之间增加一个新的分层，作为安全内核，操作系统几乎不变地作为虚拟机。安全内核的接口几乎与原有硬件等价，操作系统本身并未意识到已被安全内核控制，仍像裸机一样执行它自己的多进程和内存管理，因此它可以不变地支持现有的应用程序，且能很好地兼容非安全操作系统的将来版本。

2）改进/增强法

在现有操作系统基础上，对内核和应用程序进行面向安全策略的分析，然后加入安全机制，经改造、开发后的安全操作系统基本保持了原有非安全操作系统的用户接口。

由于改进/增强法是在现有系统的基础上增强安全性的，受其体系结构和现有应用系统的限制，因而很难达到很高（如B3级以上）的安全级别。但这种方法不破坏原有系统的体系结构，开发代价小，且能很好地保持原非安全操作系统的用户接口和系统效率。

3）仿真法

对现有操作系统的内核做面向安全策略的修改，然后在安全内核与原非安全操作系统用户接口中间再编写一层仿真程序。这样在建立安全内核时可以不必受现有应用程序的限制，且可以完全自由地定义ISOS仿真程序与安全内核之间的接口，但采用这种方法要同时设计仿真程序和安全内核，还要受顶层ISOS接口的限制。另外根据安全策略，有些ISOS的功能不安全，从而不能仿真，有些接口功能尽管安全，但仿真实现特别困难。

5. 安全操作系统的开发过程

任一系统的开发过程一般都包含以下几个步骤：

（1）系统需求分析。描述各种不同的需求。

（2）系统功能描述。准确定义应完成的功能，包括描述验证，即证明描述与需求分析相符合。

（3）系统实现。设计并建立系统，包含实现验证，即论证实现与功能描述相符合。

基于非安全操作系统开发安全操作系统的过程如图9.6所示。

6. 操作系统的安全评测

操作系统安全评测是安全操作系统实现的一个极为重要的环节，发达国家对计算机信息系统安全的可信度评测进行了长期的研究，形成了一些指导实践的原则。

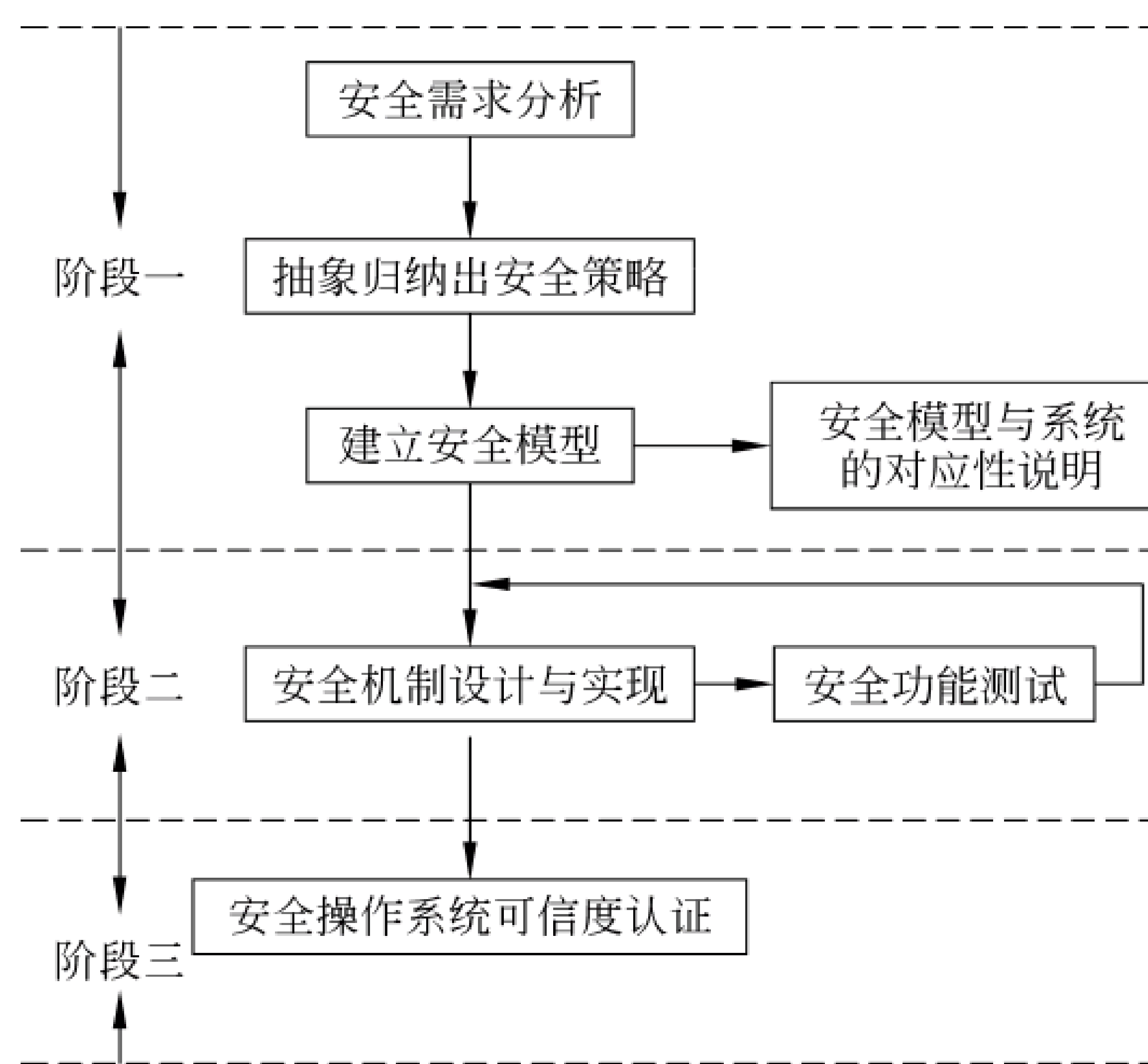


图 9.6 安全操作系统的开发过程

1）操作系统安全评测的基础

计算机安全评测的基础是需求说明，即把一个计算机系统称为“安全的”真实含义是什么。一般地说，安全系统规定安全特性，控制对信息的存取，使得只有授权的用户或代表他们工作的进程才有读、写、建立或删除信息的存取权。美国国防部早在 1983 年就基于这个基本的目标给出了可信任计算机信息系统的 6 项基本需求，其中 4 项涉及信息的存取控制，2 项涉及安全保障。

需求 1：安全策略。必须有一个显式和良好定义的安全策略由该系统实现。已知标识的主体和对象，必须有一组规则，用于确定一个已知主体能否允许存取一个指定对象。根据安全策略，计算机系统可以实施强制存取控制，有效地实现处理敏感(例如有等级的)信息的存取规则。此外，需要建立自主存取控制机制，确保只有所选择的用户或用户组才可以存取指定数据。

需求 2：标记。存取控制标签必须对应于对象。为了控制对存储在计算机中信息的存取，按照强制存取控制规则，必须合理地为每个对象加一个标签，可靠地标识该对象的敏感级，以及与可能存取该对象的主体相符的存取方式。

需求 3：标识。每个主体都必须予以标识。对信息的每次存取都必须通过系统决定。标识和授权信息必须由计算机系统安全地维护。

需求 4：审计。可信任系统必须能将与安全有关的事件记录到审计记录中。必须有能力选择所记录的审计事件，减少审计开销。审计数据必须予以保护，免遭修改、破坏或非授权访问。

需求 5：保证。为保证安全策略、标记、标识和审计这 4 个需求被正确实施，必须有某些硬件和软件实现这些功能。这组软件或硬件在典型情况下被嵌入操作系统中，并设计为以安全方式执行所赋予的任务。

需求 6：连续保护。实现这些基本需求的可信任机制必须连续保护，避免篡改和非授权改变。如果实现安全策略的基本硬件和软件机制本身易遭到非授权修改或破坏，则任何这

样的计算机系统都不能被认为是真正安全的。连续保护需求在整个计算机系统生命周期中均有意义。

以上 6 项基本需求构成计算机操作系统安全评测准则的基础。

2）操作系统安全评测方法

说一个操作系统是安全的，系指它满足某一给定的安全策略。要证明一个操作系统的安全性是与设计密切相关的，必须保证从设计者到用户都相信设计准确地表达了模型，而代码准确地表达了设计。一般评测操作系统安全性的方法有 3 种：形式化验证、非形式化确认及入侵分析。这些方法可以独立使用，也可以将它们综合起来评估操作系统的安全性。

（1）形式化验证

分析操作系统安全性最精确的方法是形式化验证。在形式化验证中，安全操作系统被简化为一个要证明的“定理”。定理断言该安全操作系统是正确的，即它提供了所应提供的安全特性，而不提供任何其他功能。但是，证明整个安全操作系统正确性的工作量是浩瀚的。另外，形式化验证也是一个复杂的过程，对于某些大的实用系统，试图描述及验证它都是根本不可能的，特别是那些在设计时并未考虑形式化验证的系统更是如此。这两个难点大大降低了有效地使用形式化验证的可能。

（2）非形式化确认

确认是比验证更为普遍的术语。它包括验证，但它也包括其他一些不太严格的让人们相信程序正确性的方法。完成一个安全操作系统的确认有如下几种不同的方法。

① 安全需求检查。通过源代码或系统运行时所表现的安全功能交叉检查操作系统的每个安全需求。其目标是，认证系统所做的每件事是否都在功能需求表中列出，这一过程有助于说明系统仅做了它应该做的每件事。但是，这一过程并不能保证系统没有做它不应该做的事情。

② 设计及代码检查。设计者及程序员在编写系统时仔细检查系统设计或代码，试图发现设计或编程错误。例如，不正确的假设、不一致的动作或错误的逻辑等。这种检查的有效性依赖于检查的严格程度。

③ 模块及系统测试。在程序开发期间，程序员或独立测试小组挑选数据检查操作系统的安全性。必须组织测试数据以便检查每条运行路线、每个条件语句、所产生的每种类型的报表以及每个变量的更改等。在这里，必须保证以一种有条不紊的方式检查所有的实体。

（3）飞虎队入侵测试

在这种方法中，飞虎队成员试图“摧毁”正在测试中的安全操作系统。飞虎队成员应当掌握操作系统典型的安全漏洞，并试图发现并利用系统中的这些安全缺陷。

操作系统在某一次入侵测试中失效，则说明它内部有错；而操作系统在某一次入侵测试中不失效，并不能保证系统中没有任何错误。入侵测试在确定错误存在方面是非常有用的。

3）操作系统安全评测的重点

一般来说，评价一个计算机系统安全性能的高低，应从如下两个方面进行：

（1）安全功能。系统具有哪些安全功能。

（2）可信性。安全功能在系统中得以实现的、可被信任的程度。通常通过文档说明测试及形式化验证说明。

7. 国内外计算机系统安全评测准则概况

美国国防部于 1983 年推出了历史上第一个计算机安全评价标准《可信计算机系统评测

准则》(Trusted Computer System Evaluation Criteria,TCSEC),又称橘皮书。TCSEC带动了国际上计算机安全评测的研究,德国、英国、加拿大、西欧等纷纷制定了各自的计算机系统评价标准。近年来,我国也制定了相应的强制性国家标准GB 17859—1999《计算机信息系统安全保护等级划分准则》和推荐标准GB/T 18336—2001《信息技术 安全技术 信息技术安全性评估准则》。表9.1给出了国内外计算机安全评价标准的概况。

表9.1 国内外计算机评价安全标准的概况

标准名称	颁布的国家或组织	颁布年份
美国 TCSEC	美国国防部	1983
美国 TCSEC 修订版	美国国防部	1985
德国标准	前西德	1988
英国标准	英国	1989
加拿大标准 V1	加拿大	1989
欧洲 ITSEC	西欧四国(英、法、荷、德)	1990
联邦标准草案(FC)	美国	1992
加拿大标准 V3	加拿大	1993
CC V1	美、英、法、荷、德、加	1996
中国军标 GJB2646—96	中国国防科学技术委员	1996
国际 CC	美、英、法、荷、德、加	1999
中国 GB 17859—1999	中国国家质量技术监督局	1999
中国 GB/T 18336—2001	中国国家质量技术监督局	2001

在各种计算机系统安全评测准则中对安全的评价级别都做了相应的说明,详细的评价级别说明在此不进行介绍,请查阅相应的安全评测准则。在各种计算机系统安全评测准则中对安全的评价级别的划分有所不同,表9.2说明了TCSEC、ITSEC、CC和中国GB 17859的评价级别的对应关系。

表9.2 TCSEC、ITSEC、CC和中国GB 17859的评价级别的对应关系

TCSEC	ITSEC	CC	GB 17859
D	E0	—	—
—	—	EAL1	—
C1	E1	EAL2	L1
C2	E2	EAL3	L2
B1	E3	EAL4	L3
B2	E4	EAL5	L4
B3	E5	EAL6	L5
A1	E6	EAL7	—

国外安全操作系统研究的新成果有SELinux和EROS。

国内现在对安全操作系统的研究也十分重视。中国科学院软件研究所、中国科学院信息安全技术工程研究所和南京大学等单位都在从事安全操作系统的研究并取得了一定的成果。SecLinux就是我国自主研究开发的安全操作系统。SecLinux是基于Linux资源自主开发的,符合第三级安全功能要求,它主要由8个部分组成:标识与鉴别、自主存取控制、强制存取控制、最小特权管理、安全审计、可信通路、密码服务和网络安全。

9.2 并行计算机操作系统

9.2.1 并行计算机系统

通常,一台计算机系统往往由一个处理器组成,其处理能力有限。为了提高计算机系统的处理能力,常常采取的有效手段是从计算机体系结构入手,采用以空间换取时间的办法,即通过增加处理部件的数量来减少处理时间。因增加处理部件途径的不同,产生了各种体系结构的并行计算机系统。按照Flynn的方法,目前有两种类型的并行计算机系统:SIMD型和MIMD型。

对于SIMD型并行计算机系统,主要有如下几种典型的体系结构:

(1) 多功能部件并行结构。把向量运算部件,如加法器、乘法器等,做成多个可同时运行的部件,对指令流中对应的指令实行并行操作,从而实现处理器内部指令级的并行。

(2) 流水线结构。把一个功能部件中的操作按节拍组织成流水过程,使得一个功能部件同时执行多条指令的流水操作,从而实现功能部件内部指令级的并行。

(3) 阵列机结构。将大量处理器按阵列排列,由统一操作系统进行管理,并由同一指令部件进行集中控制,各处理器在同一时刻执行同一指令,对数组中的不同元素进行相同的操作。这些结构所对应的并行计算机系统中,虽然各处理器可以高度并行地工作,但操作系统和指令部件仍是集中统一的,这对于具有很好的并行性的计算任务来讲效率很高,但对于并行性不是太好的计算任务而言效率较低,这主要是由于集中统一控制所制,进而出现了MIMD并行计算机系统。

MIMD并行计算机系统以分布处理器并行计算为基础,仍采用统一的全局操作系统,但增强了各分布处理器的独立控制能力,即各处理器均具有自己独立的操作系统内核,全局操作系统只需把并行任务分派给各处理器,它们即可自行管理执行。在这种结构的并行系统中,每个处理器可以比较自治地运行不同的程序,对不同的数据进行不同的操作。这种系统当前流行的有两种结构。

一种是共享内存系统,主要以对称多处理器(Symmetric Multi Processors,SMP)结构为代表,是现今最成功的并行计算机。在这种系统中,所有处理器通过高速总线或交叉开关共享同一物理内存,各处理器间交换信息和共享数据较为方便。在处理器数目不多时,系统的性能价格比较高;当处理器数目增多时,因共享内存访问速度的限制会影响系统的处理效率。近年来,SMP结构广泛应用于各种高性能网络服务器中。图9.7描述了一般SMP系统的结构。此时,多个处理器经总线连接起来,并通过总线共享主存储器。此外,每个处理器均有自己的主存Cache,用于缓存主存中的数据和指令。各处理器可以通过主存交换数

据，也可以直接通过总线交换信息。在这种系统中，所有处理器的地位都是相同的，所有资源，特别是存储器、中断及I/O设备均具有相同的可访性。

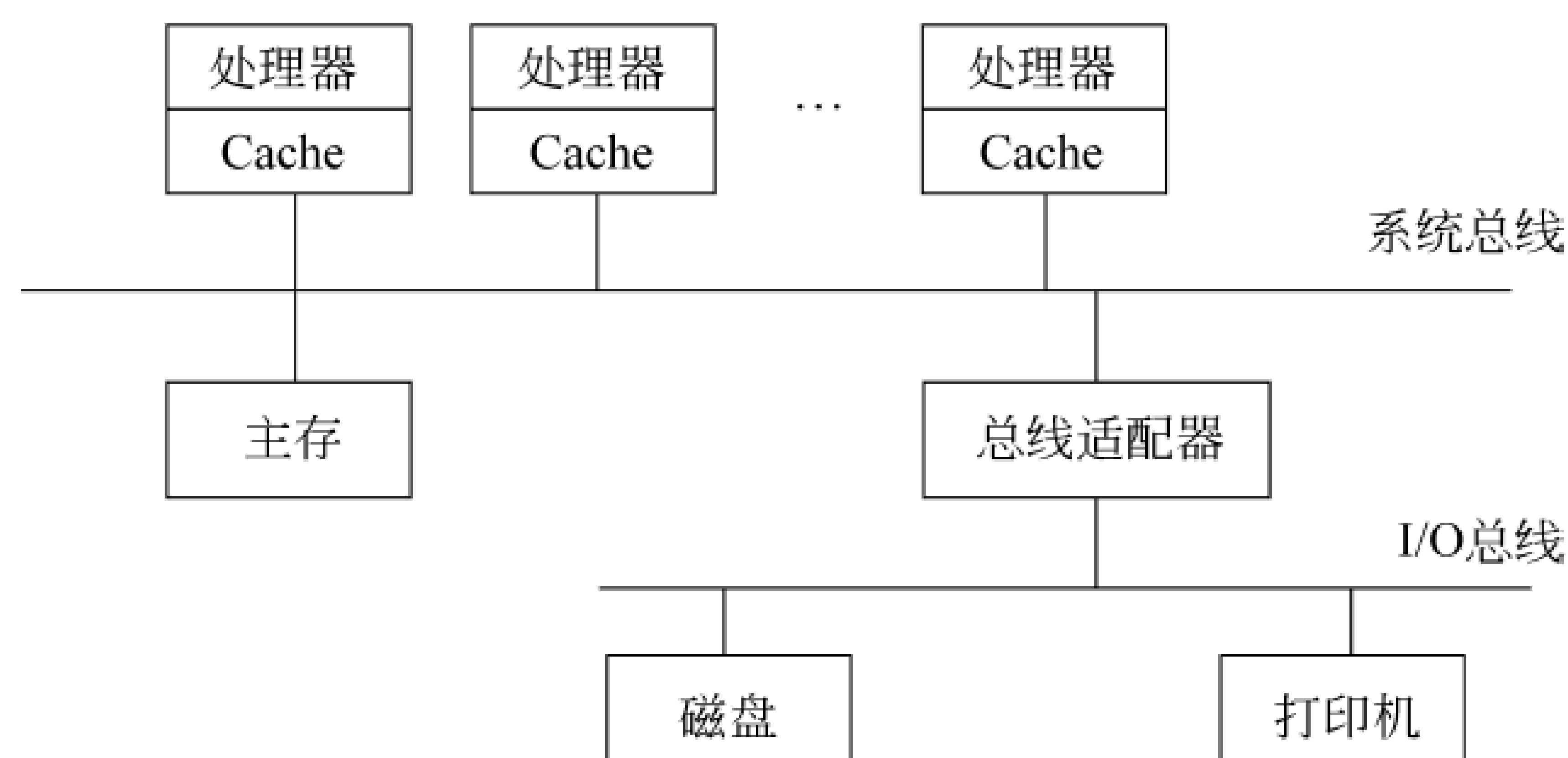


图9.7 SMP系统的结构示意图

另一种是分布式存储系统，主要以大规模并行处理(Massively Parallel Processing，MPP)结构为代表。MPP是当前超级巨型机系统流行采用的一种并行机系统结构，它将大量高性能的微处理器通过机内高速通信网络连接起来，各处理器有自己独立的操作系统内核和相当容量的内存。这种内存的分布化减少了各处理器访问共用内存时对速率的影响，致使MPP结构互连的处理器可达数万个，计算速度达10^{12}次/s以上。这类MIMD计算机系统发展很快，它支持大任务的并行执行，主要应用在科学计算、工程模拟和信号处理等以计算为主的领域。

MIMD并行计算机系统所对应的操作系统称为多处理机操作系统，它是并行操作系统中较为成熟的部分，故下面以SMP系统为例介绍多处理器操作系统中的主要技术。

9.2.2 多处理器操作系统

多处理器操作系统一般为一个分时系统。与普通的分时操作系统相比，最主要的特点是多处理器操作系统中存在单一的一个运行队列，系统内所有就绪进程在此排队。运行队列由位于共享存储器中的一个数据结构来表示，它用来记录与进程(线程)有关的状态信息。

根据各处理器可以执行操作系统的情况，多处理器操作系统一般可以分成3类：主从式、浮动式和对称式。在主从式多处理器操作系统中，只有一个处理器能执行操作系统代码，它被称为主处理器，其他处理器为从处理器。浮动式多处理器操作系统实际上也是一种主从式操作系统，但此时主处理器是变动的，即任一处理器均可能成为主处理器。但在任何时刻，只能有一个处理器为主处理器。在对称式多处理器操作系统中，任何一个处理器都可以运行操作系统代码，而且可以有多个处理器同时进入操作系统核心，这是目前最常见的多处理器操作系统。

对于对称多处理器系统，由于多个物理处理器(称为处理器池)的存在，为进程(线程)的并行执行提供了物质条件，这样多处理器操作系统就可以将多个并行进程(线程)派发到各个处理器上，做到真正地并行执行，而不是单机系统中的分时并发执行。一般来说，对称多处理器系统是相当可靠的。当一个处理器失效时，操作系统将这个处理器从处理器池中移出，并将故障通知操作人员；在出现故障处理器的修复期间，整个系统仍能在较低的级别上

继续工作(称为故障弱化)。

与对称多处理器系统相对应的操作系统应包括以下主要功能:

(1) 支持多任务并行运行。为了发挥多处理器的运行效率,要求操作系统支持多进程(线程)机制。而且多处理器可能同时执行相同的内核代码,故要求内核的例程必须是可重入的。操作系统的调度程序应能够同时调度同一进程内的各线程占用不同的处理器,以加速进程的执行过程。

(2) 具有同步机制。由于多个处理机执行的程序可能同时访问共享的地址空间或其他系统资源,所以操作系统必须提供同步机制来实现对共享资源的互斥访问。

(3) 处理器的调度和分配以及系统负载平衡。当多个并行进程(线程)需要运行时,如何为之分配处理器以较好地平衡系统负载,加快系统的处理速度也是多处理器操作系统所要解决的一个重要问题。

(4) 保证存储管理及Cache的一致性。由于各处理器拥有自己的Cache,这样系统内就存在多个Cache;而这些Cache中的内容为主存中相应内容的备份;因而对这些内容进行访问或修改时,应协调不同处理器上的页面调度机制,以保证Cache与内存的一致性。

(5) 可靠性和容错能力。当某个处理器失效时,操作系统应该能够重新组织系统,使系统仍能够保持正常运行。

多处理器系统与单机系统最大的区别是它具有多个物理处理器(包括CUP和Cache等),而处理器是最宝贵的系统资源。当有多个并行进程或线程需要运行时如何为它们分配处理器是相当重要的。因为各并行进程或线程也许要相互协作,因而就需要同步机制来协调。处理器分配得当,系统负载较为均衡,程序运行快,作业的吞吐量大;反之,可能系统负载极不均衡,各处理器忙闲不均,或者各线程因相互制约不能顺利地快速运行。下面讨论多线程系统的多处理器调度中所采取的一些方法和策略。

1. 负载共享调度

负载共享调度是指位于某一处理器就绪队列上的线程可以在任意处理器上运行,这是最简单的调度方式。采用这种调度策略的优点如下:

(1) 系统负载可以均匀地分派到各个处理器上。

(2) 处理机完全根据自身运行情况进行调度。当运行用户线程的处理器中断或自陷进入操作系统内核时,在处理完相应操作后即可转入操作系统内核调度程序,以选取相应的就绪线程投入运行。

(3) 对全局的就绪队列完全可以像在单机系统下那样进行组织和管理,可以采用单机状态下的调度策略。

在这种调度策略下,各用户的所有线程共享同一个就绪队列,各处理器在自我调度时没有目的地选择高优先级的线程。若所选择的并行线程之间需要大量同步通信,则会因同步等待而使线程频繁切换处理器,使用户作业周转时间延长。

2. 负载绑定调度

负载绑定调度是指将线程调度到指定的处理器上运行,而又允许处理器在发生I/O或同步等操作时能够进行重新调度,以切换到其他线程上运行。现代操作系统都为用户提供了将其线程绑定到特定处理器上的手段。一旦被绑定,该线程只能被调度到所绑定的处理器上运行;直到绑定被解除后,该线程才能被调度到其他处理器上运行。

3. 组调度

把一组进程同时调度到一组处理器上运行就称为组调度。其优点如下：

(1) 让一组相关的进程同时在处理器上运行，这样进程间的协作因同步通信引起的阻塞时间会缩短，减少了进程切换时间，提高了系统的运行效率。

(2) 通过将一组进程同时分派到一组处理器上运行减少了调度频度，从而减少了调度开销。

组调度已应用于进程内多线程的同时调度。利用多线程开发的中小粒度并行任务，由于线程间同步通信频繁，如果一部分线程运行，而另一部分线程不运行，则会大大降低任务的并行性。

组调度有多种形式。在共享内存的对称式多处理器系统中，采用的是浮动型组调度技术。当处理器因中断或自陷而运行操作系统核心调度程序时，它调度用户进程中的线程运行。如果用户进程要求进行组调度，且说明了处理器集合，核心调度程序就会将用户进程的所有线程分派到指定的处理器集合上去执行。此时，每个处理器均有一个待运行线程的TCB指针域。当核心调度程序进行组调度时，只需将所选进程的所有线程的TCB指针依次填入各处理器的待运行线程TCB指针域中，从而就替其他处理器完成了调度任务，然后依次向各处理器发送处理器间中断。其他处理器在进入核心调度程序时，发现其TCB指针域中已存在待运行的线程，表明已有其他处理器帮助它完成了线程调度工作，这时只需进入线程切换程序，选择一个待运行的线程占有处理器，并将之从其TCB指针域中删除。

4. 多级动态调度

现代操作系统一般都支持用户级和核心级线程。利用核心级线程，用户作业可以真正并行地在多个处理器上运行。

用户开发并行应用程序时，只希望将应用中的并行任务尽可能地挖掘出来，而无须考虑多处理器系统中可用的处理器数。用户可以利用多线库支持的用户级线程把应用中的并行任务详尽地描述出来，而这些并行任务对处理器的占用先是以用户线程的形式来表示，然后由多线库调度器将之映射到某一核心级线程上，再由操作系统内核调度到各个处理器上运行。实质上，核心级线程所运行的程序通过多线库的调度器在不同的用户并行任务程序间进行轮流转换。用户多线库中的调度程序负责将用户级线程调度到一个核心级线程上，而该核心级线程则最终由操作系统内核调度程序调度并占有处理器。由于用户级线程的调度工作全部在用户空间的多线库中完成，开销非常小。这样将用户并行任务的调度分成了两级：先用用户级线程进行用户并行任务的描述，然后在核心级上进行调度以分配处理器。

与传统的操作系统一样，核心级线程的调度发生在操作系统的核心，调度的时机是在中断进入操作系统核心程序运行即将返回时；而用户级线程的调度发生在用户空间的多线库中，调度的时机是在用户程序以任何方式调用多线程函数的时候。

近几年来，基于微处理器的SMP系统得到了普及，近而可以将这种SMP系统作为节点，组成了更为复杂的基于SMP的集群系统，实现了SMP系统节点内的并行和节点间的并行，以达到高性能计算的目的。

9.3 集群系统

随着计算机技术的迅速发展,人们对计算机性能的要求越来越高,特别是在处理庞大数据、进行大量运算的领域,最初完成这些功能的单机系统是相当昂贵的,如IBM和Cray的巨型计算机。随着计算机网络以及高速通信技术的发展,使得将多台廉价的微机、工作站以及对称多处理器系统等单台设备通过高速网络连接起来,通过软件实现任务的合理分配,使系统内的各台计算机互相协同完成同一项工作,从而构成高性能价格比的计算机系统,这种计算机系统就称为集群(clusters)系统。

由此可见,集群系统是一些功能完整的计算机(称之为节点)的集合,它们之间通过高速通信网络互相连接。这些计算机能够协同工作,对外表现为一个集成的单一计算资源,同时能十分容易扩充或缩减其计算能力。

集群系统的各节点具有协同处理能力,在用户面前应呈现为单一的计算机系统,即实现单一系统映象(Single System Image,SSI)功能,所以集群系统为分布式系统的一个较为成熟的实例。

集群系统中的节点一般有3种连接方式:

(1) 无共享方式。指节点之间通过I/O总线连接,目前大多数集群系统均采用这种连接方式。

(2) 共享磁盘方式。较小规模的商用性集群系统常常采用这种连接方式,其优点是当某个节点出现故障时,其他节点可以替代它工作。

(3) 共享存储器方式。这是一种新型的连接方式。在这种结构中,互连系统与每个节点中的存储总线相连,而在其他两种结构中,互连系统则是与各节点的I/O总线相连。

图9.8所示是一种理想的集群系统结构,其中的各个节点可以是工作站、PC、SMP服务器甚至超级计算机,各节点可以是同构的,也可以是异构的。PVM为并行虚拟机,MPI为消息传递接口,它们为用户提供并行软件编程环境。各节点的操作系统是多用户、多任务和多线程操作系统,如Linux操作系统。它们均为单机操作系统,并不提供全局服务支持,同时也缺少全局的共享方法;因此必须在节点操作系统和并行编程环境之间增加一层中间件,即所谓的集群操作系统,来解决集群系统中的资源调度、任务分配和负载平衡等问题。该中间件主要包括两部分:可用性基础设施和单一系统映象基础设施。可用性基础设施为

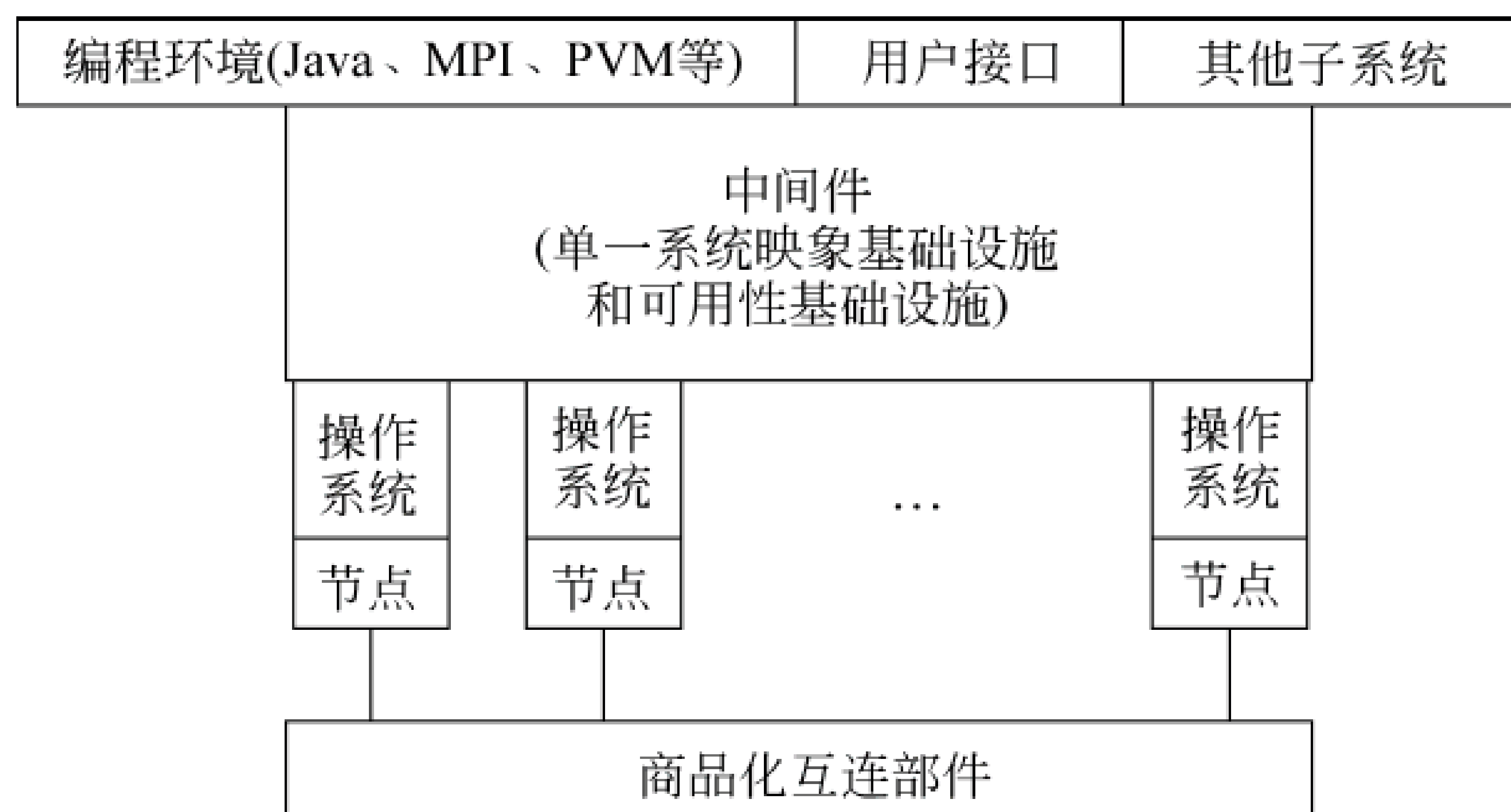


图9.8 一种理想的集群系统结构

运行在节点操作系统层之上的软件,它提供如自动故障检测、故障恢复和容错等集群服务,主要用来提高系统的可用性。单一系统映象基础设施提供单一系统映象服务功能,使得集群系统在用户面前体现为一个有机整体。

集群系统集各种同构或异构的计算机系统软、硬件资源于一身,这些系统资源的管理相当复杂,如何有效地管理系统中的资源是集群系统所要解决的一个重要问题。集群系统通过资源管理系统(Resource Management System,RMS)来对集群环境下的各种资源进行管理。RMS包括资源收集管理器和资源调度器两部分。资源收集管理器主要负责收集和管理资源,而资源调度器主要负责任务排队、资源定位和分配任务。

RMS为一种重要的中间件,是集群系统所需的一个基本配置软件,支持交互式用户登录和批处理作业,可提供一组服务用于任务分配、负载平衡和进程迁移,实现单一系统映象,解决低层平台的异构性,提供透明的资源管理等。RMS通过对资源的管理形成了集群计算环境。

集群是当前分布式并行处理研究领域的前沿技术,具有许多其他系统所不具备的优点:性能价格比高、可扩展性好、高可用性、高可靠性。尤其是它通过PC等廉价的硬件设备就可以构造出高性能的并行分布式计算系统,非常适合我国的基本国情。

9.4 分布式操作系统

分布式系统在许多方面优于独立的计算机系统,如资源共享、高可靠性、并行处理与信息通信等。分布式计算机系统是由多个分散的计算机经互联网连接而成的一个计算机系统。该系统中的各台计算机没有主、从之分,它们的地位是平等的。运行于其中任两台计算机内的进程均可通过系统提供的通信手段交换信息,而且系统中的资源为所有用户所共享,每个用户都可以透明地访问系统中其他计算机上的资源。在用户面前,分布式系统就是一个分时的单机系统,而不是一组不同的计算机,这一特性称为单系统映象(single-system image)。该系统最显著的特点是系统中各台计算机可以相互协作来完成同一项任务,或者说一个程序可以分布在几台计算机上并行地运行,这是与计算机网络系统的最大不同之处。

分布式操作系统是为分布式计算机系统所配置的操作系统,它运行在多机环境下,是负责控制和管理以协同方式工作的多类系统资源,负责分布式进程的同步与控制,完成计算机间的进程通信,进行任务分配和负载平衡,并具有高度并行性以及故障检测和自动重构能力的一种高级软件系统,它远比单机操作系统和网络操作系统复杂。

9.4.1 分布式操作系统的特点

分布式计算机系统由多台单独的计算机系统所组成,这些计算机资源是分散的,它们可以相互协作完成共同的任务,实现物理意义上的并行工作,也可以独立完成不同的工作。因而分布式操作系统远比单机操作系统复杂,它具有如下特点:

(1)进程同步机制应在一个面向消息的通信环境中进行。即常采用消息传递的通信机制,这种机制以消息队列的形式通过通信协议来实现进程间的同步。

(2)通信开销增加。分布式系统中,位于不同计算机上的进程进行通信所需时间比单机内进程间的通信所需时间要多,而且由于进行机间传输,数据出错的机会也随之增多,所

以对传输的信息要增加必要的检错和纠错措施。

(3) 资源分散,难以管理。资源管理是操作系统的主要功能之一。在分布式系统中,系统资源分布在每台计算机上,如果仍采用单机的资源管理方式,不仅开销大,系统效率低,而且系统的健壮性和可靠性均会受到影响。所以分布式系统必须采用独特的资源管理方式。

(4) 要进行负载均衡。分布式系统运行过程中,各计算机上的负载可能很不平均,有的计算机上负载很重,而有的计算机几乎长期处于空闲状态。为了充分发挥分布式系统中各组成部分的作用和效率,要协调系统各组成部分的负载情况,使其达到基本平衡,以提高分布式系统的整体效率。

(5) 系统控制的复杂性增加。在分布式系统中,各节点不存在主从关系或层次关系,因而增加了系统控制的复杂性。首先,由于各计算机的高度自治,导致它们之间发生冲突的概率很高,使同步机制变得异常复杂,死锁问题也难以处理。其次,分布式系统对于透明性要求很高,使得系统故障的检测更为困难。

(6) 系统状态的不确定性。在分布式系统中,各计算机是高度自治的,相互不报告各自的状态信息,因而很难得到对方的运行状况。另外,若系统想要收集各计算机的状态信息,由于通信的延迟,所得到的状态信息往往不能精确地反映系统的当前状态。

上述特点使得分布式操作系统设计难度增加,虽然目前已有一些商用的分布式操作系统,但功能完整的分布式操作系统尚处于实验阶段,仍属于分布式系统研究的前沿领域。

9.4.2 分布式操作系统的构成

分布式操作系统大都采用微内核结构,它由微内核和各功能模块所组成。微内核仅提供支持系统运行的最基本功能,包括进程通信、低级网络通信、进程调度和中断处理等。系统的其他功能模块包括一组系统服务进程,用于完成内核之外的各种功能,如文件服务、目录服务、虚拟存储管理和设备驱动等。这种微内核结构隐藏了硬件上的异构性,提供了统一的内核接口,使得分布式操作系统能够适应不同的硬件平台,从而使得分布式操作系统可以运行在各种同构和异构的硬件环境下。

分布式操作系统无论运行在哪种系统环境下,它都是系统中全部计算机所共享的一个公共操作系统。系统中的各台计算机未必安装分布式操作系统的所有部分,操作系统的各个模块可以不均匀地分布在分布式系统内的各台计算机上。但是所有计算机上均装有操作系统内核的副本,该内核对计算机系统进行基本的控制。

除具备单机操作系统的各种功能外,分布式操作系统提供计算机间的进程通信和资源共享能力,并为用户提供透明地访问其他计算机上资源的方法。

9.4.3 分布式操作系统的通信

分布式操作系统中的多台计算机是相互独立、自治的,它们可能需要相互交换信息以协作完成共同的任务。下面先介绍一种最重要的分布式计算模型:客户/服务器模型(Client/Server, C/S),然后介绍常用的两种分布式操作系统的通信方式:消息传递和远程过程调用。

1. 客户/服务器计算模型

客户/服务器计算是目前广泛采用的分布式计算模型,它由客户机和服务器所组成。通

常客户端是单用户 PC 或工作站,它为用户提供使用界面。服务器的配置一般较高,它提供一系列的共享服务供客户端使用。目前最常用的服务器为数据库服务器,该服务器使得众多客户端共享同一数据库,并为各客户端提供高性能的计算机系统来管理数据库。该计算模型包括一对并行执行的进程:客户进程与服务器进程,它们之间通过通信进行交互作用,共同协作来完成同一项任务。此时,客户进程发送请求给服务器进程,服务器进程接受客户端的请求,然后进行响应和处理,并将结果返回给客户端。客户/服务器的体系结构如图 9.9 所示。

在客户/服务器计算模型中,客户端和服务器分别起到不同的作用。根据所承担的主要处理工作的不同,将该计算模型分成基于主机、基于服务器、基于客户端和客户/服务器协同处理 4 种。

2. 分布式消息传递

在分布式操作系统中,协作进程一般采用消息传递机制来实现不同计算机上进程之间的数据通信。

图 9.10 是一种最普通的用于分布式消息传递的通信模型,也是一种客户/服务器模型。一个客户进程请求一些服务(如读文件),向服务器进程发送一个请求服务的消息。服务器进程接收请求并做出应答。采用两个函数就可以简单地实现上述过程:发送函数 Send()和接收函数 Receive(),这两个函数可以通过调用原语来实现。

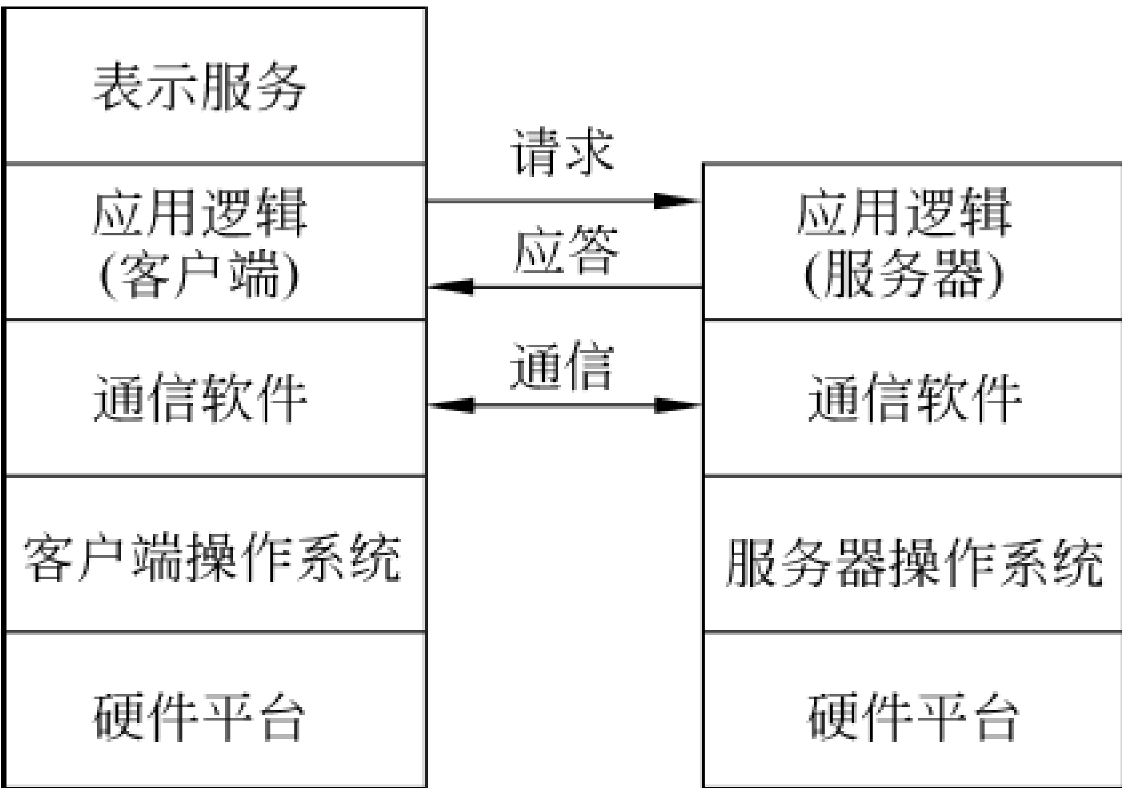


图 9.9 客户/服务器的体系结构

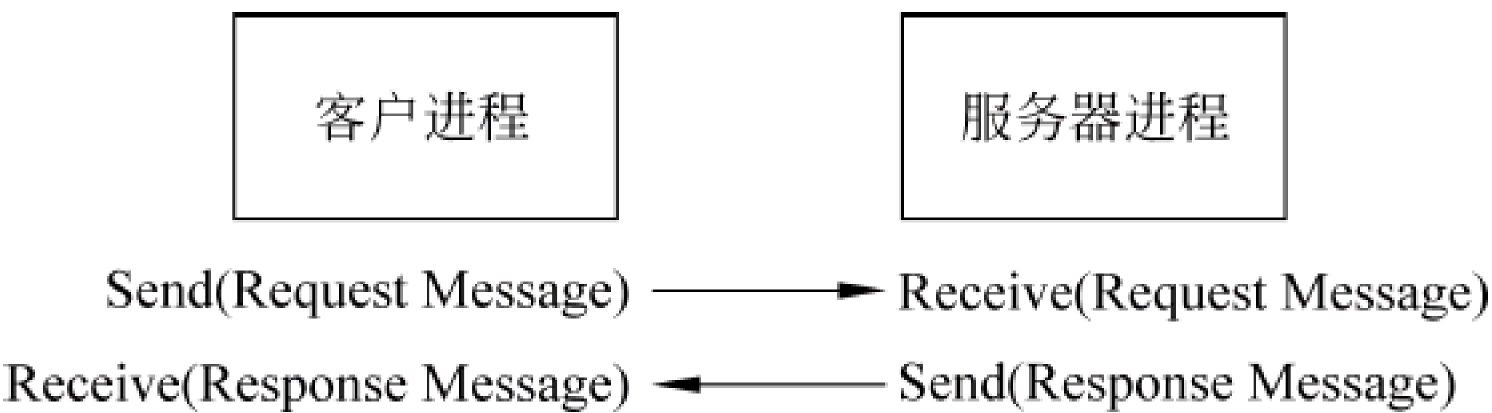


图 9.10 基本的消息传递模式

3. 远程过程调用

目前分布式操作系统中广为应用的另一种通信方式是在基本消息传递模块的基础上加上远程过程调用(Remote Procedure Calls,RPC)。这种方式允许一个站点上的程序调用其他计算机上的过程,该过程接收请求并进行处理,然后将结果通过参数表传送给调用的程序;对于用户来讲,该过程执行起来就好像执行本地的过程调用一样。

远程过程调用机制可以看作是一种可靠的分块消息传递,如图 9.11 所示。调用程序在客户端完成带有参数的普通调用,如

$$\text{CALL } P(X,Y)$$

其中 P 为过程名,X 为传递的参数,Y 为返回值。这个调用将产生一个包括有关被调用过程标识和参数的消息。然后客户本地存根(stub)程序将这个消息打包,通过内核发送到远程服务器系统,并等待响应。发送完消息后,客户本地存根程序调用接收原语,然后阻塞直到收到服务器发来的应答。

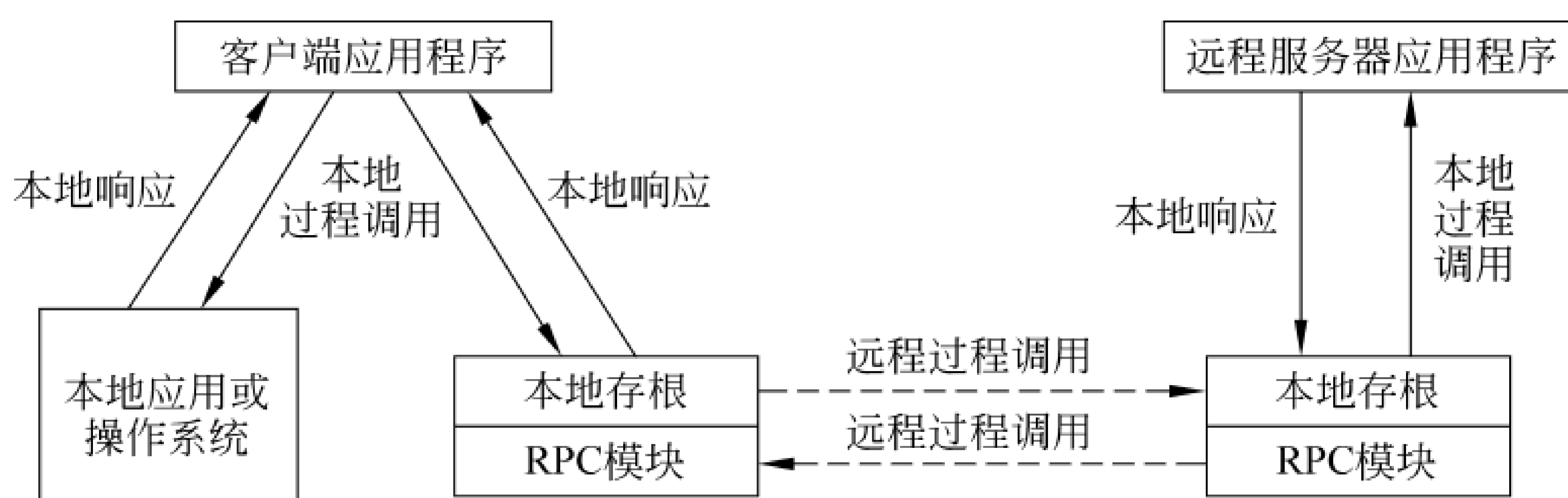


图 9.11 远程过程调用机制

当消息到达远程服务器系统后,内核将消息传送给与实际服务器进程相捆绑的服务器本地存根程序,服务器本地存根程序从中取出参数,并产生一个本地调用 CALL $P(X,Y)$,要求服务器提供服务。服务器接收请求,并将处理结果放在服务器本地存根程序的缓冲区内。当服务器端的本地调用完成后,服务器本地存根程序获得控制权,然后将结果(缓冲区的内容)打包,通过调用发送原语将消息返回给客户端。

当消息返回到客户端并被复制到等待缓冲区中,客户存根程序检查并拆开消息包,取出结果,并将它复制到调用者进程的缓冲区中。

9.4.4 分布式操作系统的资源管理

资源管理是操作系统的一项重要任务。分布式系统由于资源的分散性和多样性,致使其资源的管理方式不同于单机系统,否则系统的开销大而且健壮性差。

分布式操作系统的资源管理具体有两种方式:

(1) 分布式集中管理方式。指一类资源由多个管理者来管理,但每个具体资源只存在一个管理者。具体地说,资源按其分布情况分别由其所在的站点实现局部的集中管理,不存在全系统范围的集中管理者。例如,分布式系统中存在多个存储器,各自分布在相应的计算机内,这类资源有多个管理者,但每台计算机内的存储器仅有一个管理者,即它自身。显然,这种管理方式下,资源管理者对它所管理的资源具有完全的控制权。因为每个资源具有唯一的管理者,故该资源的管理类似于单机操作系统,但增加了通信手段以申请使用其他站点上的资源。

由于各站点都具有隶属于自身管理的一部分资源,所以进程请求资源时首先申请自己站点上的资源,只有当该请求得不到满足时才转去申请其他站点上的资源。

(2) 分散管理方式。指一类资源由多个管理者共同协商管理。这种管理方式下,资源管理者对于所管理的资源仅有部分控制权,所以实现起来较为复杂。此时,由于分布式系统内各计算机的自治性,各计算机既能管理分布于本地的资源,又能接收和处理其他计算机对其本地资源的请求。由于各功能模块的多个副本往往分散在多台计算机上,但又不能采取全系统的集中控制策略,所以必须注意对系统资源管理的一致性。这种管理方式实现复杂,通信开销大,要求采用较好的算法来提高对资源请求的响应速度。

在分布式操作系统的实现过程中,根据资源的重要程度,采用分级的管理方式,即大多数资源采用分布式集中的方式进行管理,仅对经常被各个站点使用的资源即共享资源采用分散管理方式。

9.4.5 分布式进程管理

在分布式操作系统中，由称为进程服务器的进程管理模块管理计算机资源，实现进程的创建、终止和迁移等操作。有两种计算机管理模型：计算机池模型和本地节点模型。

在计算机池模型中，进程服务器管理一个计算机池供全局使用。它保存系统中各计算机的负载信息。当一个新作业到达时，它根据系统负载情况，从计算机池中为该作业分配一台计算机。

本地节点模型主要基于如下思想：用户大多数情况是登录到本地计算机，因此总是隐含地将用户进程放到本地节点上运行。在该模型中，通常要实现进程迁移机制，以便当某个节点负载过重或出现故障时，进程服务器能够通过进程迁移机制将进程迁移到负载较轻或可靠的节点上，而且这种负载平衡或进程迁移的过程对用户是透明的。

在分布式操作系统中，负载平衡只是进程迁移的一个原因。此外，进程迁移还有如下作用：

（1）减少通信量，提高通信性能。

（2）利用特定的系统资源。利用进程迁移可以使进程移动到更有利于它执行的站点上去运行，从而可以充分利用某一特定节点上的软、硬件资源，提高进程的运行速度。

（3）增强系统的可用性。当进程所在的计算机出现故障时，该进程可以迁移到运行正常的计算机上继续运行。

进程迁移是一个非常复杂的过程，它涉及迁移的原因、迁移的内容和迁移的事后处理等。

9.4.6 分布式进程的同步、互斥与死锁

如同单机操作系统一样，分布式环境下的进程管理也需考虑进程的同步、互斥和死锁问题，但此时的情况要比单机环境复杂得多。因为在分布式环境下，每个节点只是系统的一部分，决策仅局限于本地信息。另一方面，在单机环境下，常常按照先来先服务的原则根据进程请求的次序为进程分配临界资源。但在分布式环境下，系统没有公共时钟，每个节点都有自己的局部物理时钟，这些时钟不可能完全一致，故很难确定各个进程发出资源请求的次序。此外，不同节点上的进程通过网络进行通信，而由于网络延迟致使难以判断多个资源请求的先后关系。所有这些因素导致在分布式环境下实现进程之间的同步和互斥是相当复杂和困难的。

为了解决分布式环境下进程的同步/互斥问题，Lamport 首先提出了时钟同步的可能性，并引入了逻辑时钟的概念。逻辑时钟是一个单调递增的软件计数器，用来标识事件发生的相对顺序，而不是事件发生的绝对时间。每个系统均采用自己的逻辑时钟作为它所产生事件的时间戳。对 Lamport 算法感兴趣的读者可以阅读有关的参考文献。

除 Lamport 算法外，也可以借鉴计算机环形网中共享介质的令牌访问控制方式，实现分布式系统中进程对临界资源的互斥访问。

死锁是分布式系统中的另一个难于解决的问题，而且远比单机系统复杂。通常也采用类似于单机系统的死锁解决方案，即忽略、预防、避免和检测死锁。对于分布式系统来讲，要实现死锁的避免和预防是很困难的，所以一般采用的方法是忽略死锁或者采用死锁的检测

和恢复。感兴趣的读者可以参考相应的高级操作系统教程。

9.4.7 分布式文件系统

该文件系统同单机操作系统中的文件系统一样,用于管理磁盘空间和提供文件服务等,为用户提供一个单一的、全局的、对用户透明的文件系统。该文件系统也称为文件服务器,一般运行在一个配备大容量磁盘的专用服务器上。当客户程序需要访问一个文件时,它向服务器发出请求并得到返回结果。在分布式系统中,允许有多个文件服务器,它们之间必须协调才能保证系统为用户提供一个单一的逻辑文件系统,从而对用户屏蔽了物理服务器的数目和位置。为了提高系统的性能和可靠性,分布式文件系统自动实现文件重复存放和文件迁移,而且整个过程对用户是透明的。

9.5 网络操作系统

计算机被誉为20世纪人类社会最伟大的发明之一,它短短几十年的历史却给世界带来了翻天覆地的变化。从某种意义上讲,计算机网络与通信技术的发展水平已成为一个国家综合实力和科学技术发展程度的重要标志。

网络操作系统(Network Operating System,NOS)是计算机网络的重要组成部分,像单机操作系统一样,它是网络中各种软、硬件资源的管理者,是网络用户和网络系统间的接口,它为用户屏蔽了本地资源和网络资源的差异,为用户提供透明的网络服务功能,在很大程度上它决定了网络系统的性能。至20世纪90年代,网络操作系统已趋于成熟。

9.5.1 计算机网络简介

计算机网络是指将分散在不同地理位置、具有独立功能的计算机、终端以及其他通信设备用通信线路连接起来,按照一定方式进行通信,并实现资源共享的系统。

综观计算机网络的发展历史,也是由简单到复杂、由单一的机间通信发展到Internet上的资源共享、功能逐渐完善的过程。计算机网络的分类有多种方法,按照规模划分可以分成局限于某一地理范围内的局域网(Local Area Network,LAN),可以覆盖一个城市或地区的城域网(Metropolitan Area Network,MAN),可以覆盖一个国家甚至全球的广域网(Wide Area Network,WAN);而按照用途或性质划分可以分成企业网、校园网或专用网、公用网,等等。常见的局域网有以太网、令牌环网和FDDI网等,广域网有帧中继、ISDN和ATM等。

计算机网络一般由网络软件和网络硬件两部分组成。网络软件用于对系统资源进行协调、管理以及合理地调度和分配,实现对交换信息的发送和接收控制,并采取一系列的保密措施,防止非法用户访问系统资源或合法用户滥用其特权,以确保网络系统正常、可靠地工作,所以网络软件是实现网络功能所不可缺少的软环境。通常的网络软件包括以下几类:

(1) 网络协议和协议软件。它通过协议程序实现网络协议功能。网络协议规定了通信双方传输数据时所必须遵守的语义、语法和时序规则,以便能相互识别对方所发送来的信息。

(2) 网络通信软件。通过该软件实现网络系统中不同站点之间的通信。

(3) 网络操作系统。管理各种软、硬件资源,控制用户对网络资源的访问,实现资源共享功能,为其他软件提供运行环境和低层支持,它是最主要的网络软件。

(4) 网络管理及网络应用软件。网络管理软件用来对网络资源进行管理和维护,如流量计费、安全管理和设备状态监测等。网络应用软件为网络用户提供各种服务,如 WWW 服务、E-mail 服务和远程登录等。

网络硬件是计算机网络运行的基础,它由节点和通信介质组成。节点包括计算机、路由器和交换机等通信设备,通信介质包括光纤、同轴电缆、双绞线和无线电波等。

计算机网络的主要目的是实现资源共享。随着网络技术的发展,其应用范围和应用领域越来越广。

9.5.2 计算机网络体系结构与协议

目前,存在两种标准的网络体系结构,一种是由国际标准化组织公布的开放式系统互连参考模型 ISO/OSI(Open System Interconnection),另一种是工业标准的 TCP/IP(Transport Control Protocol/Internet Protocol)网络参考模型。其中,开放式系统互连参考模型共分 7 层:物理层、数据链路层、网络层、传输层、会话层、表示层和应用层,各层功能如下:

(1) 物理层(physical layer)。物理层为通信提供物理链路,实现比特流的透明传输。物理层定义了与通信介质以及接口硬件有关的机械、电气、功能和过程的各种特性和标准,以便建立、维护和拆除物理连接。

(2) 数据链路层(data link layer)。数据链路层用于提供相邻节点间透明、可靠的信息传输服务。该层传输的基本数据单位称为帧。

(3) 网络层(network layer)。网络层用于提供源端和目的端间的信息传输服务。该层提供路由选择、差错控制和流量控制等。该层还向传输层提供数据报或虚电路服务。该层传输的基本数据单位称为分组(packet)。

(4) 传输层(transport layer)。传输层为通信双方提供端到端的、可靠的数据传输,执行端到端的差错控制、顺序和流量控制,实现多路复用等功能。该层传输的基本数据单位称为报文。

(5) 会话层(session layer)。会话层为通信双方建立会话联系,使它们按同步方式交换数据,并有序地拆除连接,以保证不丢失数据。

(6) 表示层(presentation layer)。表示层向应用层提供信息表示方式,并对不同的表示方式进行转换。该层完成的功能有数据压缩、加密和格式转换等。

(7) 应用层(application layer)。应用层为用户进程访问 OSI 环境提供手段,直接为应用程序提供服务,各低层均通过该层为应用进程提供服务。

上述 7 层协议中,物理层和数据链路层的功能通常在网络接口卡中实现,而其他 5 层协议的功能则在网络操作系统中实现,其中网络层和传输层等的功能在通信过程中所起的作用尤其关键,故一般在网络操作系统的内核中实现,以保证具有较高的执行优先级和通信可靠性。

TCP/IP 网络参考模型是随着以太网的广泛应用而得到工业各界普遍认可的一种标准,它比开放式系统互连参考模型(ISO/OSI)出现得要早。该网络参考模型共分 4 层:通信子网层、互连层、传输层和应用层。各层功能如下:

(1) 通信子网层(subnet)。通信子网层是TCP/IP网络参考模型的最低层。它接收互连层送来的IP报文,然后进行处理并通过物理网络将该IP报文发送出去。或者从物理网络接收物理帧数据,解析出IP报文,然后将之递交给网络层。该层通常执行其固有的通信协议。该层相当于开放式系统互连参考模型(ISO/OSI)的最低两层。

(2) 互连层(internet layer)。互连层也称IP层,主要实现网络互连功能,它可以实现多种异种网络的互连。该层提供路由选择、报文分段和地址解析等功能。该层的主要协议为IP协议,所传输的基本数据单位称为IP报文。

(3) 传输层(transport layer)。传输层为通信双方提供端到端的、可靠的数据传输服务,执行端到端的差错控制、流量控制、数据分段和多路复用等功能;通过超时重传等机制确保数据可靠地传输,而通过滑动窗口机制实现网络中数据传输流量的控制。该层包括TCP(传输控制协议)和UDP(用户数据报协议)等协议。

(4) 应用层(application layer)。应用层为TCP/IP网络参考模型的最高层,它为用户使用网络的各种功能提供服务,包括远程登录服务协议(Telnet)、文件传输协议(File Transfer Protocol,FTP)和简单邮件传输协议(Simple Mail Transfer Protocol,SMTP)等。远程登录协议允许用户登录到远程系统并能够像远程计算机的本地用户一样访问远程系统的各种资源。文件传输协议为两台计算机之间传输数据提供手段。简单邮件传输协议最初只是文件传输的一种类型,后来发展成一种特殊的服务,即电子邮件业务。随着计算机网络技术的成熟以及应用领域的扩大,应用层又新增了许多协议,如用于将网络中的主机名字地址映射成网络地址的域名服务(Domain Name Service,DNS),用于传输网络新闻的NNTP(Network News Transfer Protocol)和用于支持WWW服务的超文本传输协议(Hyper Text Transfer Protocol,HTTP)等。

目前,基于TCP/IP网络参考模型的Internet已遍布全球,构筑了人们生活、学习和工作的信息平台,已成为现代社会不可分割的一部分。

9.5.3 网络操作系统的发展及分类

随着计算机网络应用的普及,网络操作系统也发生了很大变化,逐渐从对等结构演变成非对等结构,由以共享硬盘服务为中心发展到以共享文件服务为中心的、功能强大的网络操作系统。

1. 对等结构的网络操作系统

在对等结构操作系统中,所有网络节点的地位是平等的,安装在每个节点上的操作系统软件相同,各节点上的资源原则上都可以相互共享。这种结构的网络操作系统可以提供共享硬盘和共享打印机等功能,具有结构简单、网络中的任何节点均能直接通信等优点。

对等结构的网络操作系统因每个节点要完成工作站和服务器的双重职能,不但要承担本地用户的信息处理任务,还要完成较重的网络通信管理和共享资源管理,因而工作负荷较重,致使其信息处理能力明显降低。所以,对等结构网络操作系统支持的网络规模较小。

2. 非对等结构的网络操作系统

为了克服对等结构网络操作系统的缺点,提出了非对等结构的网络操作系统。此时网络节点分成两类:网络服务器(server)和网络工作站(workstation),它们分别承担不同的工作。网络服务器一般采用高配置和高性能的计算机,以集中方式管理网络资源,并为网络工

作站提供各种服务。网络工作站一般是配置较低的微机系统，主要用于管理本地资源，并为访问网络资源提供手段。

该种结构的网络操作系统软件分成协同工作的两部分，一部分安装并工作在服务器上，另一部分安装在工作站上。网络服务器集中管理网络资源和服务，是计算机网络的核心。它所使用的网络操作系统则直接决定网络服务功能的强弱以及系统的性能和安全性。

在早期的非对等结构的网络操作系统中，通常在网络中配备一台或几台大容量硬盘作为供多个网络工作站用户使用的共享磁盘服务器，以便为网络用户提供服务，所提供的服务有：共享磁盘、共享打印机、电子邮件及通信服务。这种共享磁盘服务器系统在用户使用前均需要建立与服务器的连接，用户需要自己建立和维护所访问的文件目录结构，因此使用起来很不方便，系统效率低，安全性差。

3. 基于文件服务的网络操作系统

为了克服非对等结构网络操作系统的缺点，提出了基于文件服务的网络操作系统。这类网络操作系统分成以下两部分：文件服务器和工作站软件。其中，文件服务器具有分时系统文件管理的全部功能，它支持文件的概念和标准的文件操作，为网络用户提供访问文件、目录的并发控制和安全保密措施。所以，文件服务器一般均具备完善的文件管理功能，能够对全网实行统一的文件管理，并为网络用户提供完善的数据、文件和目录服务。

目前的网络操作系统都属于这类系统，如 Microsoft 公司的 Windows NT 和 Novell 公司的 NetWare 操作系统等。这些网络操作系统提供了强大的网络服务功能和优越的网络性能，为计算机网络的广泛应用奠定了基础。

9.5.4 网络操作系统的功能

随着计算机网络应用的普及，对用于管理计算机网络的操作系统的功能的要求也越来越高，促使网络操作系统的功能逐渐增强，由最初的数据通信、资源共享发展到网络管理、安全管理、计费、流量监测、路由选择和故障检测等，这些功能分别用于完成网络体系结构模型中各层协议所规定的功能。由于计算机网络由计算机以及其他通信设备所组成，所以单机操作系统应为网络操作系统的一个真子集。网络操作系统还具有单机操作系统所不具备的另外一些功能，如数据通信、网络管理、用户访问控制和海量数据文件管理等。

1. 数据通信功能

数据通信是计算机网络最基本的功能。为了在不同的计算机之间实现数据通信，网络操作系统应该具有如下功能：

(1) 建立和拆除连接。在计算机网络中为了使源主机和目的主机进行通信，必须建立两个主机间的连接。当数据通信结束后要拆除连接，以释放所占用的通信资源。

(2) 控制数据传输。为了能够控制数据的正确传输，常常将一些控制信息作为头部和被传输的数据组成分段或报文后一起传输。

(3) 检测差错。数据在网络中传输时难免出现错误，因而网络必须提供差错控制措施。

(4) 流量控制。由于网络中各节点的通信能力并不一定完全一致。有的节点发送/接收数据的速度快，而有的节点速度慢。若不对流量进行控制，速度慢的节点有可能来不及接收数据而造成数据丢失。因而网络协议软件要考虑流量的控制。

(5) 选择路由。计算机网络进行传输数据时可能存在许多路径。当中间节点接收到一

个分组数据时,它要根据自己对网络连接情况的了解,并按照一定的策略(如传输路径最短、传输延迟最短等)为转发的数据选择一个输出路径。

(6) 多路复用。计算机网络中必须提供多路复用功能,以提高物理线路的利用率。所谓多路复用是指将一条物理链路虚拟成多条虚电路,每条虚电路为一对用户使用,这样就允许多对用户同时使用一条物理链路进行通信。

2. 网络管理的功能

组成计算机网络系统的设备数量多,而且分散,难以管理。所以只能借助网络协议软件来自动地进行管理,其目的在于提高网络系统的利用率,最大限度地增加网络系统的可用时间,改善网络的服务质量,保证网络系统的安全运行等。

国际标准化组织为网络管理定义了网络配置、性能管理、故障管理、安全管理和计费管理等五大功能。

(1) 配置管理。配置管理涉及定义、收集、监视、控制以及使用网络系统的配置数据。配置数据包括网络中各种重要资源的静态和动态信息。配置管理用于监视网络中的各种配置数据,允许网络管理人员生成、修改、优化和查询网络软硬件资源的运行参数和状态,以保证网络工作在正常的状态下。

(2) 性能管理。通过收集网络各组成部分运行情况的有关信息,从而分析网络的运行情况,如网络的响应时间、数据吞吐量和网络的阻塞情况,对网络的工作状态进行评价,并可将评价的结果作为网络配置效果的反馈信息。

(3) 故障管理。故障管理通常用于检测网络在运行过程中所发生的异常现象,以发现故障,然后根据故障情况对故障进行跟踪、诊断、测试、定位和修复等,并将故障情况通知网络管理员,以进行必要的人工干预,同时还要在系统日志文件上进行记录。

(4) 安全管理。根据安全策略来控制对受限资源的访问。在安全管理中,常用的方法和技术有认证技术、访问控制技术、数据加密技术、密钥分配与管理、安全日记的维护和检查、审计和跟踪、防火墙和入侵检测等。这些方法有的由网络操作系统实现,有的往往需要网络操作系统内核的支持,有的则需要人机协同来完成。

(5) 计费管理。计费管理用于监视和记录用户使用网络资源的种类、时间和数量,必要时可以调整用户使用网络资源的配额,并对用户所分配到的网络资源的使用情况进行计费。计费管理所涉及的具体功能有收集计费记录、计算用户账单、检查资费变更情况和分配网络运行成本等。

3. 网络文件系统

计算机网络为用户所提供的功能之一是信息共享,而这些信息一般以文件的形式分布在不同的计算机上。这些计算机系统往往是异构的,所使用的操作系统多种多样,所以网络环境下所共享文件的格式存在很大的差异。为了方便用户对这些信息的存取,网络操作系统中提供了网络文件系统(Network File System,NFS)来实现异构环境下的文件管理功能。它为用户提供一个统一接口,使得不同操作系统平台上的用户共享同一个文件系统。这样,通过 NFS 将用户的本地文件系统和网络上的多个远程文件系统结合成一个整体,构成了一个虚拟的文件系统,并呈现在用户面前,使得用户感觉不到访问远程文件系统与访问本地文件系统的区别。从而 NFS 对用户屏蔽了异构环境下文件系统的多样性,为上层用户提供了统一的、虚拟的文件操作平台,在下层通过不同的接口与具体的文件系统进行交互。

VFS(Virtual File System)/VNODE(Virtual File Node)接口就是上述技术的一个具体实现,VFS/VNODE的结构如图9.12所示。

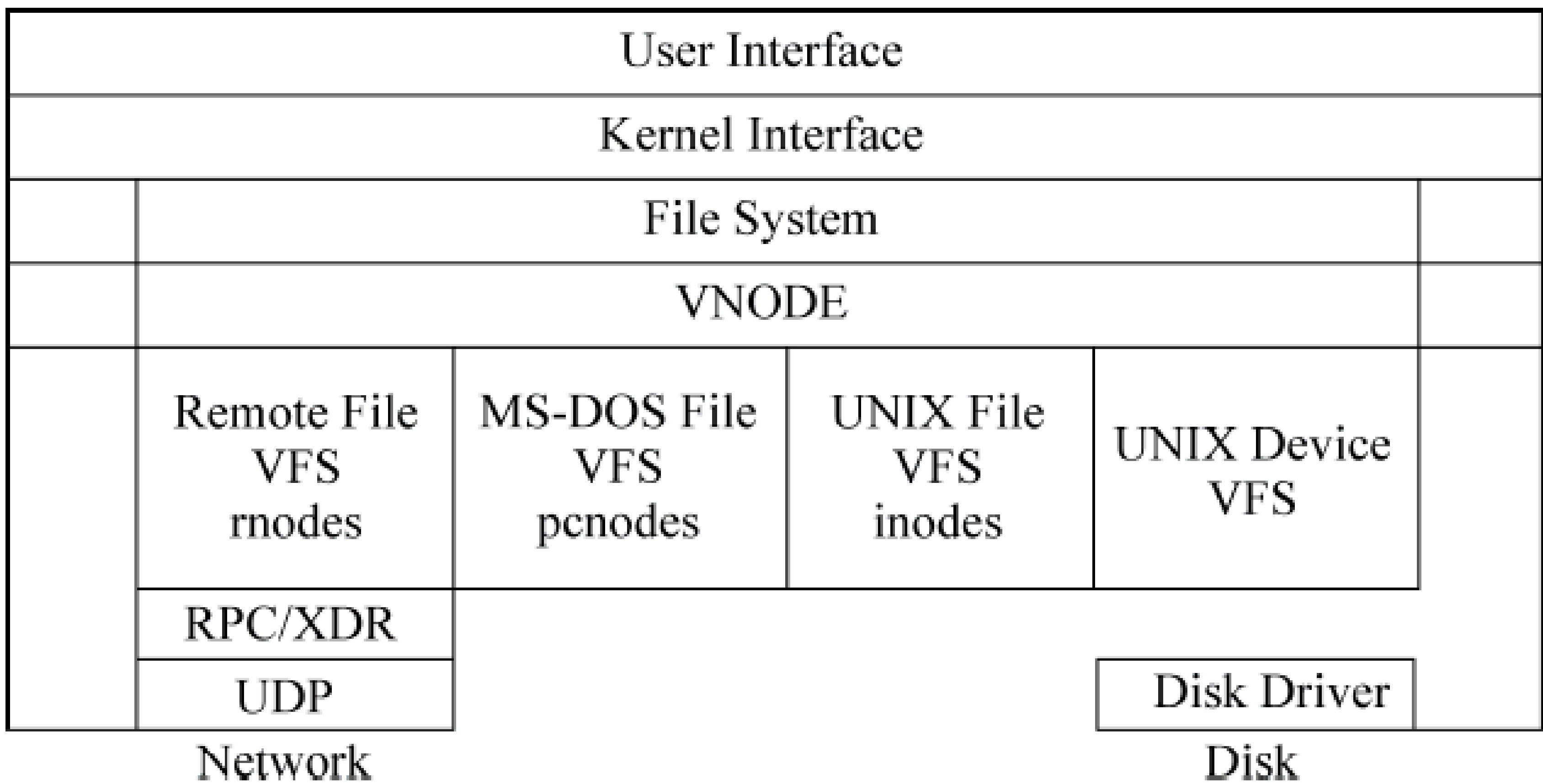


图9.12 VFS/VNODE的体系结构

9.5.5 网络操作系统提供的服务

为了方便用户使用网络,网络操作系统都提供了一系列有效的服务功能,其中包括文件传输服务、电子邮件服务、目录服务以及Internet上的WWW服务等。

文件传输服务是一种十分有用的信息服务。网络用户可以利用该服务来实现对远程网络主机或服务器的访问,存取远程文件系统中的文件,执行文件传送操作等。传输的文件可以是文本文件、二进制可执行文件以及多媒体文件等。

目录服务实现对网络三大资源——物理设备、网络服务和网络用户的有效管理,远比单机环境要复杂得多。对于物理设备而言,目录服务要为它们建立一张目录表,表中每个目录项记录物理设备的标识符、设备类型和设备的物理地址等。对于工作站要记录所配备的操作系统类型,对于服务器要记录所能提供的服务类型。目录服务也把用户作为管理对象,通过目录来记录用户的有关信息,如用户名、用户口令、允许用户访问的计算机名、用户账户的有效时间等。由于目录管理的对象相当庞大,在目录库中的目录项就非常多,如果将它们集中放在一个目录服务器中,则会导致该服务器的负担过重,影响其工作效率,所以一般将庞大的目录库分成若干个分区,分别复制到地理位置上与经常访问这些目录的用户相近的服务器上,并允许一台目录服务器存放多个不同分区的副本。这样目录服务就还需要完成目录库的分区和复制等维护、管理工作。

电子邮件服务为用户提供电子邮件的编辑、发送、接收、分发和安全保护等功能。而WWW服务为用户提供信息的发布、浏览和检索等服务,使得用户可以共享由Internet所提供的庞大的信息资源。

9.6 多媒体操作系统

数字电影、视频和音频成为一种用一台计算机代表信息和娱乐的潮流。音频和视频文件存储在磁盘上,按要求播放。然而它们的特性与当前被设计成存放传统文本文件的系统有很多不同之处。由此需要一种新的文件系统来处理它们。更重要的是,存储和播放视频

和音频数据间存在一定的联系,这对操作系统提出了新的特殊要求。本章讨论与这些有关的关键性问题以及在以多媒体为处理对象的操作系统上它们是如何实现的。

通常数字电影被称为多媒体,多媒体字面的含义是多于一种媒体。按这种定义方式,本书可认为是一本多媒体书籍。它包含两种媒体:文本和图像(数字)。然而大多数人用“多媒体”表示包含两种或两种以上媒体,它们必须以一定的时间间隔连续播放。

支持多媒体的操作系统与传统的操作系统有所不同,主要在3个方面:处理器调度、文件系统和磁盘调度。

9.6.1 多媒体引入

在单台计算机上,多媒体经常表示从DVD(数字通用磁盘)播放一个事先录制的电影。

多媒体的一个应用是通过Internet下载视频片。许多Web页面有这样的条目,点击它可以下载短的电影。

多媒体应用的另一个重要领域是电脑游戏。

多媒体领域中最重要的是视频点播(Video On Demand,VOD),通过它人们体验到了在家中利用电视机的遥控器或鼠标来选择电影,使电影在电视机或计算机的显示器上即时播放。为了能点播,需要一种特殊的基础结构,它包含3个必需的部件:一个或多个视频服务器和一个分布式网络,以及为了解压信息在每个房间里要有的机顶盒。视频服务器是一个功能强大的计算机,在它的文件系统中存储许多部电影,可按要求播放它们。有时大型计算机被用作服务器,把用来存放电影的上千个大型磁盘连接到大型机上。

在用户和服务器之间的分布式网络必须具有实时高速传输数据的能力。如果想更多地了解这种网络的设计请参阅相关资料。

系统的最后一个部分是机顶盒,它是一个带有为视频解码和解压芯片的一般的计算机。

作为机顶盒的替代品,可用用户已有的PC,在PC的显示器上播放电影。设置机顶盒的主要原因是让人们想在已有电视但没有PC的卧室里看电影。

多媒体有以下两个关键的特性:

(1) 媒体数据量庞大。

(2) 多媒体要求实时播放。

提高服务质量的最常用的方法是对每一个新用户事先预留资源。预留的资源包括CPU、内存区域、磁盘传输容量和网络带宽等。如果一个新用户到来,想看电影,但用户得不到视频服务器等足够的网络资源。为了避免降低处理当前用户的服务质量,服务器不得不拒绝新用户。因此,多媒体服务器需要资源预留和接受控制方法来决定它们能处理多少工作。

9.6.2 多媒体文件及视频压缩

1. 多媒体文件

在多媒体系统中包含许多不同结构的文件。视频信息和声频信息是完全不同的,它们被不同的设备(CCD片和麦克风)所捕获,具有不同的内部结构(视频有25~30帧/秒,音频有44.100样本/秒),它们通过不同的设备(显示器和声音播放器)播放。

另外,许多电影是在不同的语言地区播放,这些地区的观众可能不懂电影的原声语言。

解决这一问题有两种方法：一种方法是通过增加一个声道，用当地的语言进行配音，当然仅仅是声音效果；另一种方法是为电影提供原声语言的字幕。因而一部数字电影包含许多文件：一个视频文件、多个声频文件和多个语言字幕文本文件。DVD 有保存 32 种语言和字幕文件的能力。图 9.13 显示的是一部电影包含多种文件的例子。

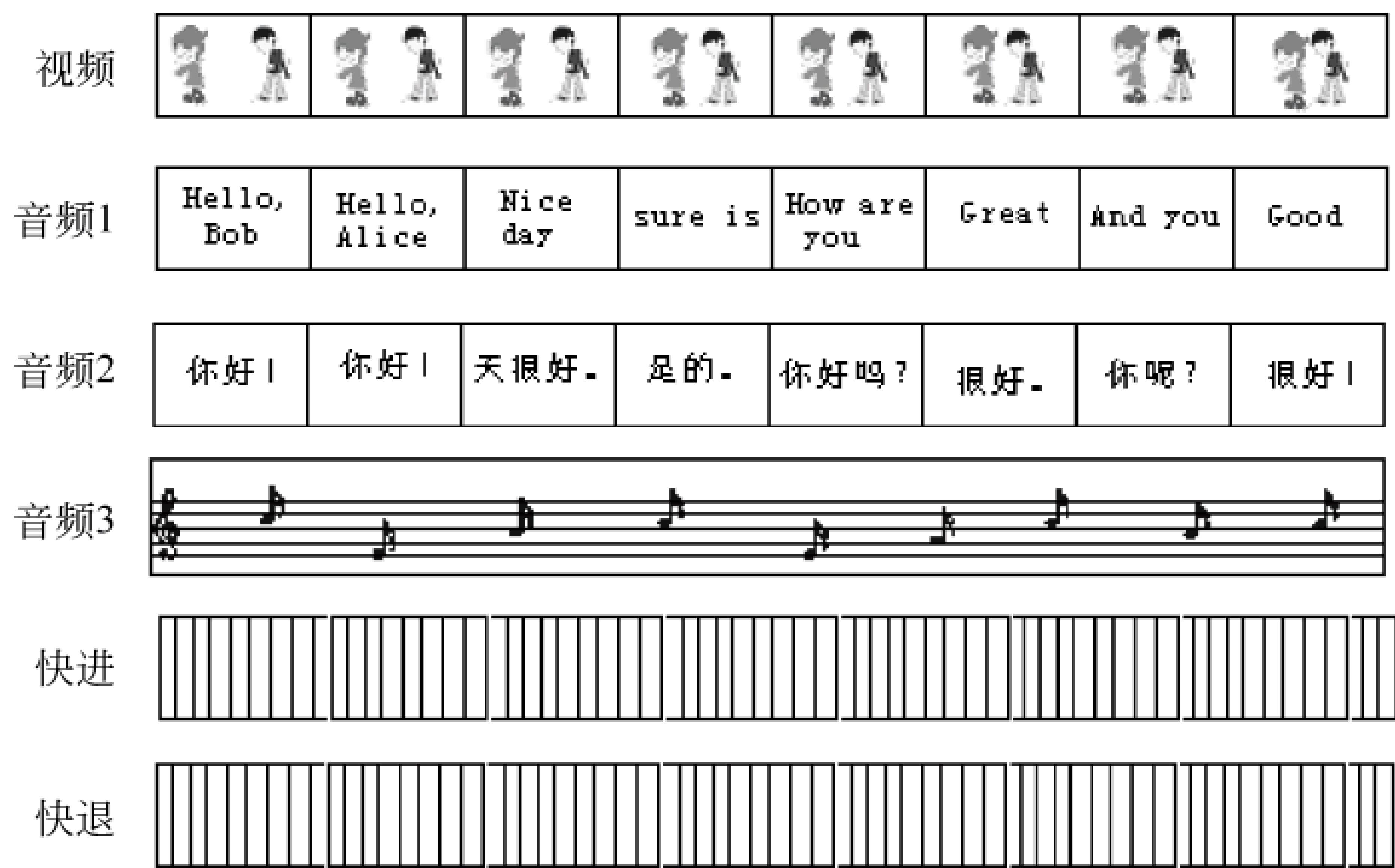


图 9.13 一部电影包含多种文件

因此，文件系统需要保持每一个文件的多个子文件索引。

在任何情况下，保持子文件同步的方法都是非常必要的，当选定声道播放时，它必须与视频保持同步。

2. 视频压缩

因为多媒体文件太大，所以以非压缩形式处理多媒体是不可能的。被压缩的多媒体文件在播放时必须被解压，因此要求有两个算法：对原始数据进行压缩的算法和对目标数据进行解压的算法。

1）常用的数据压缩技术

自 1948 年 Oliver 提出脉冲编码调制(Pulse Coding Modulation，PCM)编码理论以后，人们已经研究了各种各样的多媒体数据压缩方法，从不同角度对数据压缩方法进行分类。

根据解码后数据与原数据是否完全一致，可将数据压缩方法分为两类：

(1) 可逆编码(无失真编码)。如 Huffman 编码、算术编码和行程长度编码等。

(2) 不可逆编码(有失真编码)。常用的方法有变换编码和预测编码。

根据压缩的原理可以将数据压缩方法分为以下 6 类：

(1) 预测编码。它是利用空间中相邻数据的相关性，利用过去和现在出现的点的数据情况来预测未来点的数据，常用的方法有差分脉冲编码调制和自适应差分脉冲编码调制。

(2) 变换编码。该方法将图像光强矩阵(时域信号)变换到频域空间上进行处理。在时域空间上具有强相关的信号，反映在频域上是某种特定的区域内能量常常被集中在一起，人们只需将主要精力集中在相对小的区域上，从而实现压缩。一般采用正交变换，如离散余弦变换、离散傅里叶变换、Walsh-Hadamard 变换和小波变换来实现压缩算法。

(3) 量化与向量量化编码。对模拟信号进行数字化时，要经历一个量化过程。为了使整体量化过程失真最小，就必须依据统计的概率分布设计最优的量化器。

(4) 信息熵编码。这是根据信息熵原理,让出现概率大的符号用短的码字表达,出现概率小的符号用长的码字表达。最常用的方法有 Huffman 编码、算术编码和 Shannon 编码等。

(5) 子带编码。将图像数据变换到频域后,按频域分带,然后用不同的量化器进行量化,从而达到最优组合。也可以采用分步渐近编码,在初始时,对某一频带的信号进行解码,然后逐渐扩展到所有频带。随着解码数据的增加,解码图像也逐渐变得清晰。

(6) 模型编码。编码时首先将图像中的边界、轮廓和纹理等结构特征找出来,然后保存这些参数信息。解码时根据结构和参数信息进行合成,恢复原图像。具体方法有轮廓编码、域分割编码、分析合成编码、识别合成编码、基于知识的编码和分形编码等。

2) 图像的压缩标准

(1) 静态图像压缩标准 JPEG

ISO/IEC 10918 号标准《多灰度连续色调静态图像压缩编码》即 JPEG 标准选定 ADCT(自适应离散余弦变换)作为静态图像压缩的标准化算法。

常用算法有:

① 基于 DPCM 的无失真编码;

② 基于 DCT 的有失真压缩编码。

(2) 运动图像压缩标准 MPEG

MPEG 标准是面向运动图像压缩的一个系列标准。最初 MPEG 专家组的工作项目是 3 个,即在 1.5Mb/s、10Mb/s、40Mb/s 传输速率下对图像编码,分别命名为 MPEG-1、MPEG-2 和 MPEG-3。1992 年,MPEG-2 适用范围扩大到 HDTV(高清晰度电视),能支持 MPEG-3 的所有功能,因而 MPEG-3 被取消。同时为了满足不同的要求,又陆续增加了其他一些标准,如 MPEG-4、MPEG-7 和 MPEG-21。

在 MPEG 中将帧分为 3 种类型:I 帧(intra picture)、P 帧(predicted picture)和 B 帧(bidirectional picture)。

I 帧是利用帧自身的相关性压缩,提供压缩数据流中的随机存取的点,采用基于 ADCT 的编码技术,压缩后每个图素为 1~2b。

P 帧使用最近的前一个 I 帧(或 P 帧)预测编码得到(前向预测),也可以作为下一个预测的参照帧。

B 帧在预测时,既可以使用前一个帧作为参照,也可以使用后一个帧作为参照(向后预测)或同时使用前后两个帧作为参照(双向预测)。

9.6.3 多媒体处理调度

在一些实时系统中,进程具有优先级。在多媒体系统中,进程一般存在优先级,这就意味着贻误了时限处于危险中的进程可能在运行着的进程处理完它的帧之前而被中断。具有优先级的实时调度算法在多媒体系统中是适用的,它们比没有优先级的算法提供了更好的效果。这里唯一需要关注的问题是如果传输缓冲区偶然被全部占有,到时限时缓冲区是满的,此时缓冲区中的数据仍要被传送给用户,否则可能引入抖动。

具有优先级的实时调度算法分为静态的和动态的,静态算法事先分配给每个进程一个优先权,动态算法没有固定优先权。

1. 多媒体处理调度常用算法简介

在多媒体系统中常用的处理器调度算法是频率单调调度算法和时限调度算法。因为在第 4 章中对这两种算法进行了介绍,在此只作简单说明。

对于具有优先级的周期性的进程来说,典型的静态实时调度算法是频率单调调度算法(Rate Monotonic Scheduling,RMS),由 Liu 和 Layland 于 1993 年提出,他们还证明在静态调度中 RMS 是最优的。

时限调度算法(Earliest Deadline First,EDF)是另一个实时调度算法。EDF 是动态算法,它不要求进程是周期的,也不要求每次运行需要相同的 CPU 时间,但每次进程需要声明 CPU 时间、出现时间和结束时间(称为时限),调度进程保持一个能运行的进程序列,该序列是按时限排序的,算法调度序列中的第一个进程是具有最近时限的进程,无论何时一个新进程成为就绪,系统都查看它的时限是不是在当前运行进程的时限之前,如果是,新进程从当前进程中抢占 CPU。

EDF 对任何能调度的进程序列都可以使用,它可使 CPU 100%使用,代价是使用一个复杂的调度算法。在一个实际的以频率为主的调度中,如果 CPU 的利用低于 RMS 算法的限制,RMS 算法能被使用,否则使用 EDF 算法。

2. 调度相同参数的进程

视频服务器最简单的类型是能够支持一定数量的电影播放,并且具有相同的帧率、视频分辨率、数据传输率和其他参数。在这种环境下,一个简单但有效的调度算法如下:每个电影对应一个进程(或线程),它一次从磁盘上读一帧,然后把它传给用户。所有的进程具有相同的优先级,对每一帧做相同的工作,轮转调度恰好适合完成这一工作。对标准调度算法需要增加的是分时机制来保证每一个进程以相同的频率运行。

执行恰当分时的一种处理方法是设置一个主时钟,每秒钟发出 30 次时钟脉冲(对 NTSC 系统),每一脉冲使所有进程以相同的次序顺序执行。当一个进程完成它的工作,就发一个中断请求,释放 CPU 直到主时钟发出下一个时钟脉冲。当时钟发出脉冲时,所有的进程又以相同的次序运行。只要进程数量足够少,使得所有的工作在规定的时间内完成,轮转调度是有效的。

9.6.4 多媒体文件系统

多媒体文件系统使用了与传统文件系统不同的方式。在传统文件的 I/O 操作过程中,为了访问一个文件,进程首先发出一个 OPEN 系统调用。如果成功,先返回文件标识给访问者,在以后的访问中被使用。然后可以发出 READ 系统调用,参数是文件标识、缓冲区始址和字节个数。接着操作系统把要求的数据从磁盘中读出,传送到缓冲区中。之后可以进行下一次读操作,直到过程结束。最后,发出 CLOSE 系统调用,关闭文件,归还所占资源。

对于多媒体系统来说,实时行为有特殊的要求,为了显示从远处的视频服务器上传来的多媒体文件,利用上述方式工作的结果会特别糟。一个问题是用户必须发出一个在时间上特别精确的读访问。第二个问题是视频服务器必须能够无延迟地提供数据,在无计划的请求到达和事先没有安排资源的情况下视频服务器做这件事是困难的。

为了解决这些问题,多媒体文件服务器(也称视频服务器)使用了一个完全不同的文件系统方式。为了读一个多媒体文件,用户进程发出一个 START 系统调用,说明对文件的操

作是读和其他各种参数,然后视频服务器按要求的速率传输帧。用户按它们输入的速率进行处理。如果用户想终止这部电影,则发出一个 STOP 系统调用终结这个流。带有这种流式的文件服务通常称为推服务(push service),与传统的拉服务(用户必须以块为单位重复使用 read 得到数据)相对,两者之间的区别在图 9.14 中说明。

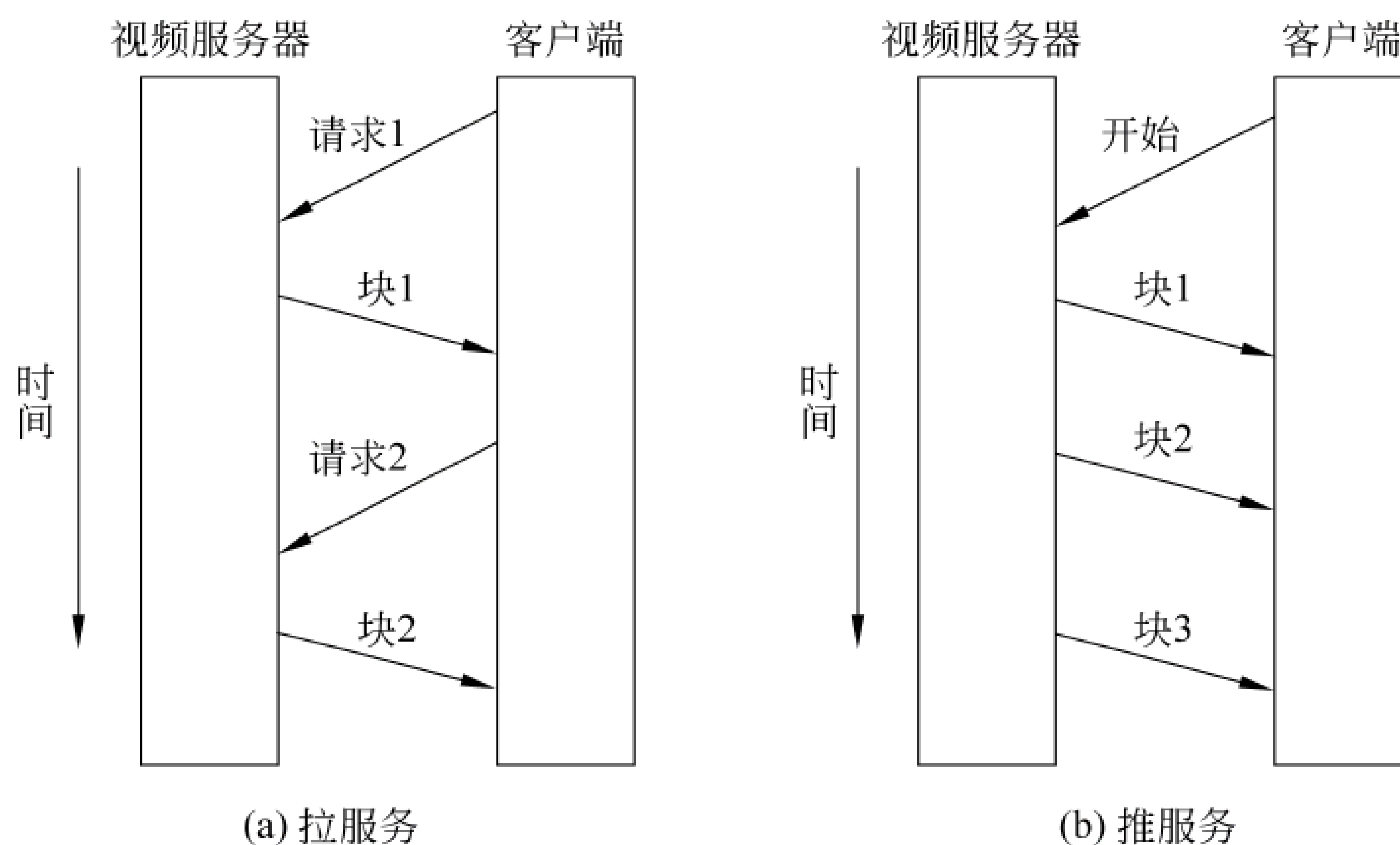


图 9.14　拉服务和推服务的比较

1. VCR 控制功能

大多数视频服务器实现了标准 VCR 控制功能,包括暂停、快进、快退、前跳和后跳。暂停是直接发出,用户发一个消息到视频服务器告诉它停止,同时它还得记住下面是哪一帧;当用户告诉服务器继续时,它恰好能从停止的位置继续播放。

然而事实并不是这样简单,而是相当复杂的。为了达到可接受的效果,视频服务器必须为每一个发出暂停请求的进程预留一些资源,如磁盘的带宽、内存储器中的缓冲区等。当一部电影暂停时,继续占用预留的这些资源是一种浪费。如果用户打算离开稍长一段时间则更是浪费。通过按暂停键,资源很容易被释放,当然这样用户再继续时,不能重新继续,从而将引入危险。

真正的回退是很容易的,服务器必须做的是标记被传输的下一帧是 0 帧。快进和快退(例如反绕时播放)是比较复杂的。如果文件不是压缩的,以 10 倍速度向前的一种方法是间隔 10 帧显示。事实上在不压缩的情况下,以任何速度快进或快退都是容易的,对于以通常速度的 K 倍快进,恰好间隔 K 帧显示;以通常速度的 K 倍快退则向反方向做同样的事情。这种方法对于推服务和拉服务都是相同的。

压缩使得快速移动无论采用哪种方法都比较复杂,对于 Camcorder DV 带,每一帧都被独立压缩,用这种对策使提供需要的帧能够很快地被找到成为可能。虽然每一帧依据它的内容被压缩成大小不同的块,在文件内快进 K 帧不能通过记录字节数量来进行。更进一步地,音频压缩独立于视频压缩,所以每一视频帧以高速度显示,正确的音频帧也必须被定位。否则当播放速度比通常快的时候音频文件将关闭。快进一个 DV 文件要求一个索引,它允许帧被快速定位,至少在理论上可以做到。

对于 MPEG,这样的安排即使在理论上也是不可行的,因为使用了 I 帧、P 帧和 B 帧。向前跳 K 帧(假设可以这样做)可能定位在 P 帧,而该 P 帧是以 I 帧为基础,I 帧被跳过了,

没有基本帧，P帧所包含的内容就是没有用的，MPEG要求文件被顺序播放。

解决这一问题的另一个方法是以10倍速度顺序播放文件。然而这样做要求以10倍速度从磁盘中取数据，那样服务器将试图解压这些帧（通常它不做这些事情），确定哪一帧是需要的，再把每10帧重新压缩为1帧。这样做把大量的下载物放在服务器中，它也要求服务器理解压缩的格式，通常它不必了解。

通过网络实际传输所有的数据给用户和让正确的帧被选中两者交替进行，要求以10倍速度进行网络传输，以网络通常必须运作的给定的高速度进行，也许可能做到，但不容易。

总而言之，没有一种容易的方式，唯一可行的策略是要求预先安排。必须做的是要建立一个特殊的文件，用通常的MPEG算法以每10帧压缩为1帧的压缩率压缩这一文件。这一文件如图9.13中显示的“快进”文件。为选择一个快进模式，服务器必须做的是指出用户当前在快进文件中的位置。例如，如果当前帧是48210，文件以10倍速度快速向前，服务器必须将快进文件定位在4821，以常规的速度在此开始播放。当然那一帧可以是P帧或B帧，但在客户端进程可以跳过若干帧，直到找到I帧为止。为了实现快退，以相似的方法准备第二个特殊文件。

当用户切回正常速度，必须做索引转换。如果在快进文件中当前帧是5734，服务器恰好回到普通文件，在57340帧上继续。如果两者当前帧不是I帧，客户端的解码程序将忽略一些帧，直到找到I帧。

即使增加两个额外的文件（快进文件和快退文件）来做这些工作，该方法也存一些缺点。首先需要一些额外的磁盘空间来存放额外的文件；其次，快进和快退仅能以与特殊文件相符的速度进行；第三，在普通文件、快进文件和快退文件之间选择向前、向后增加了复杂性。

2. 准点播

使 K 个用户看到相同的电影，必须把相同的负载加载到服务器上，就像使用户得到 K 个不同的影片一样。然而如果在处理上做些小的变化，将可能获得最大效能。点播的问题是在任意时刻用户都能看电影，所以如果有100个用户大约在晚上8点钟全都开始看相同的电影，最极端的可能是没有两个用户在完全相同的时刻开始，因此他们不能拥有同一个输出流。为了实现最优化，一种可行的做法是告诉用户电影只能在整点和每间隔5分钟开始。这样如果一个用户想在8:02看电影，他只能等到8:05。

这里假定电影播放时间是两个小时，那么无论有多少个用户，仅需要120/5＝24个流。如图9.15所示，第一个流在8:00开始，当第一个流在9000帧时，第2个流开始；在8:10，第一个流在18000帧，第2个流在9000帧，第3个流开始；……所有的流都是以0帧开始和结束。这个安排称为准点播（near video on demand）这是因为视频没有按要求尽快开始，而是在短时间内开始。

这里的关键是流是按怎样的频率开始的。如果每2分钟开始一个流，对于两个小时的电影需要60个流，而开始看电影的最大等待时间是2分钟。操作者必须决定人们愿意等待多长时间，因为他们愿意等待的时间越长，系统越有效率，同时有更多的电影可以看到。

从直观上看，点播很像打的士：你招它，它就来。准点播很像乘公交车：它有固定的调度表，你必须等着下一辆。利用准点播，用户没有VCR控制，这样用户不能暂停电影，之后再从暂停出处继续看，而是调整到另一个流上，这个流能与他刚才看的衔接上，因而可能有些重复。

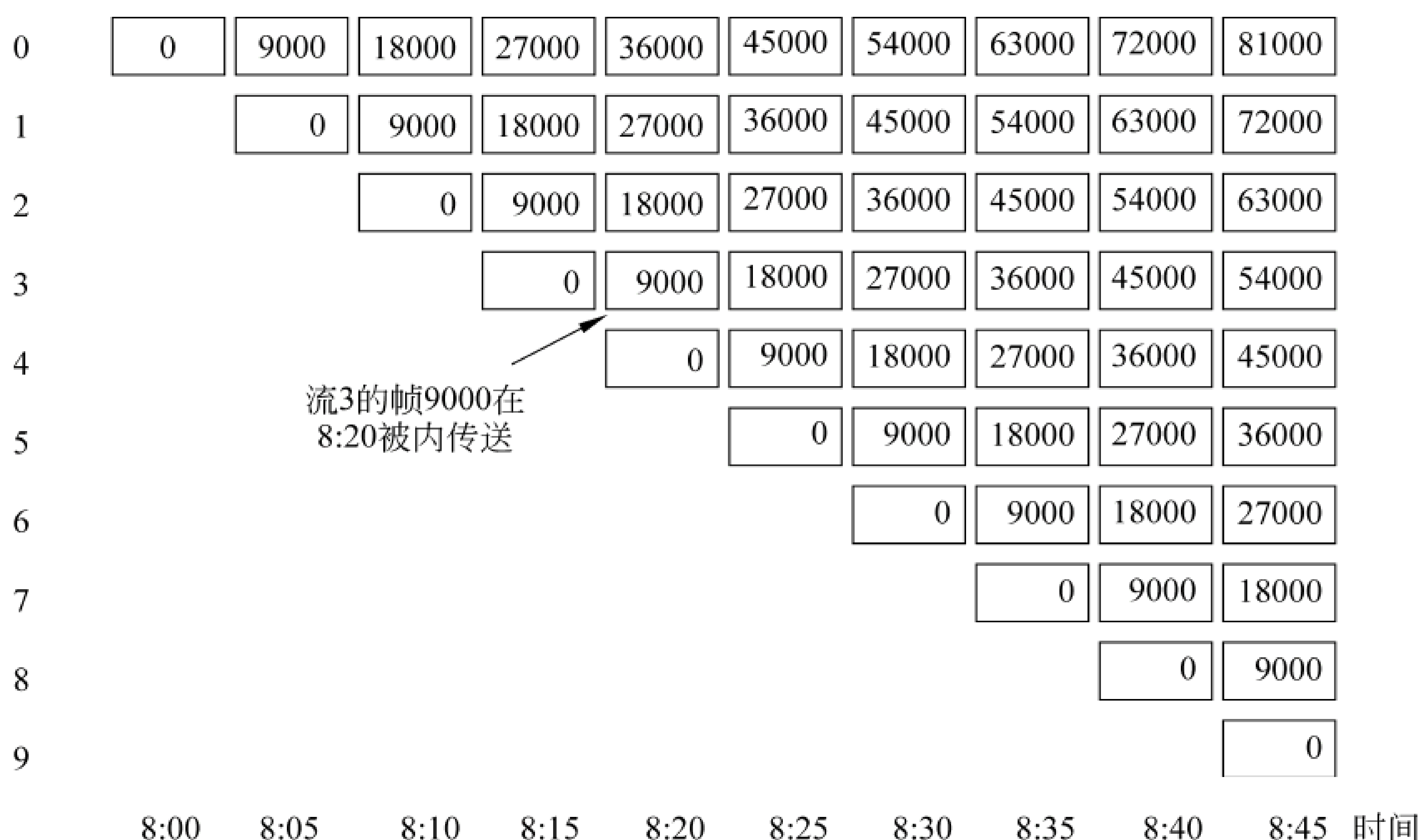


图 9.15　准点播以一定的时间间隔开始一个新的流，本例为间隔 5 分钟（9000 帧）

另一种准点播不是声明间隔若干分钟开始播放一次电影，而是用户预约电影。无论用户什么时间想看电影，间隔一定时间，例如 5 分钟，系统查看哪些电影被预订，它就开始播放哪些电影。利用这种方法，电影可以在 8:00、8:10、8:15 和 8:25 开始，但不能立刻开始。需要说明的两点是：一方面没有观众点播的电影将不被传输，节省磁盘带宽、内存和网络容量；另一方面，在观众的流开始 5 分钟之后存在另一个流是没有保证的，进行暂停是有点风险的。当然操作者能提供一种选择：用户可以显示所有并发的流，但大多数人认为，他们的电视遥控器已经有太多按钮，不喜欢再增加额外的按钮。

3. 带有 VCR 功能的准点播

典型的结合是准点播（对于效率）加上每个观众拥有 VCR 控制。对处理稍加改变，这样的设计是可能的。下面给出这一功能的一种实现方法，该方法是 Abram-Profeta 和 Shin 于 1998 年提出的。

人们用图 9.15 所示的标准的准点播调度开始，然而增加了要求：每个用户的计算机缓存前 ΔT 时间和即将到来的 ΔT 时间的帧，缓存前 ΔT 时间帧是容易的，而缓冲即将到来的 ΔT 时间的帧比较困难，但如果有用户同时读两个流是能做到的。

用一个例子来说明填充或更新缓冲区内容的一种方法：如果一个用户在 8:15 开始看电影，用户的计算机读和显示 8:15 的流（流从 0 帧开始），同时它读和存储 8:10 的流，这个流当前在 5 分钟标记处（例如 9000 帧）。在 8:20 从 0 帧到 17999 帧被存储在缓冲区中，用户想看接下来的 9000 帧。这样，8:15 的流被调整，缓冲区内存储 8:10 的流（拥有 18000 帧），从缓冲区的中间点（9000 帧）开始显示。每读出一帧，在缓冲区的末端添加一个新帧，一个帧被调整为缓冲区的始端，被显示的当前帧称为播放点，总是缓冲区中间的帧。电影播放到 75 分钟时的状态如图 9.16(a)所示。这时所有 70～80 分钟的帧都在缓冲区中。如果数据率是 4Mb/s，一个存储 10 分钟电影的缓冲区要求 300MB 的存储容量。按当前的价值，缓冲区最好建立在磁盘上，也可建立在 RAM 中。如果建立在 RAM 中，300MB 太多，可以建立一个小的缓冲区。

现在假设用户决定快进或快退。只要播放点在70～80分钟之间,显示可以从缓冲区中的内容开始。然而如果播放点的内容不在缓冲区中,问题出现了。解决的方法是转到一个私人的流(例如点播所对应的流)上为用户提供服务。在任何一个方向上的快速移动可以通过前面讨论过的技术来处理。

通常在某一点上用户要停下来,决定按正常速度播放。在这一点上考虑把用户并入准点播流中的一个流上,这样私人流被调整。例如,假设用户决定返回到12分钟的标记上,如图9.16(b)所示,这一点在缓冲区之外,所以显示不能从缓冲区中的内容开始。更进一步,即使切换发生在75分钟,播放5、10、15和20分钟的流存在,但没有一个流在12分钟。

解决的方法是继续在私人流上观看,但是从当前15分钟的流进入电影来开始填充缓冲区。3分钟后状态如图9.16(c)所示。现在播放点是15分钟,缓冲区中包含15～18分钟的帧,准点播的流是在8、13、18和23分钟。在这点上,私人流被调整,可以从缓冲区中的内容开始播放,利用这一私人流继续填充缓冲区。下一播放点是16分钟,缓冲区包括15～19分钟,输入缓冲区的流在19分钟,如图9.16(d)所示。

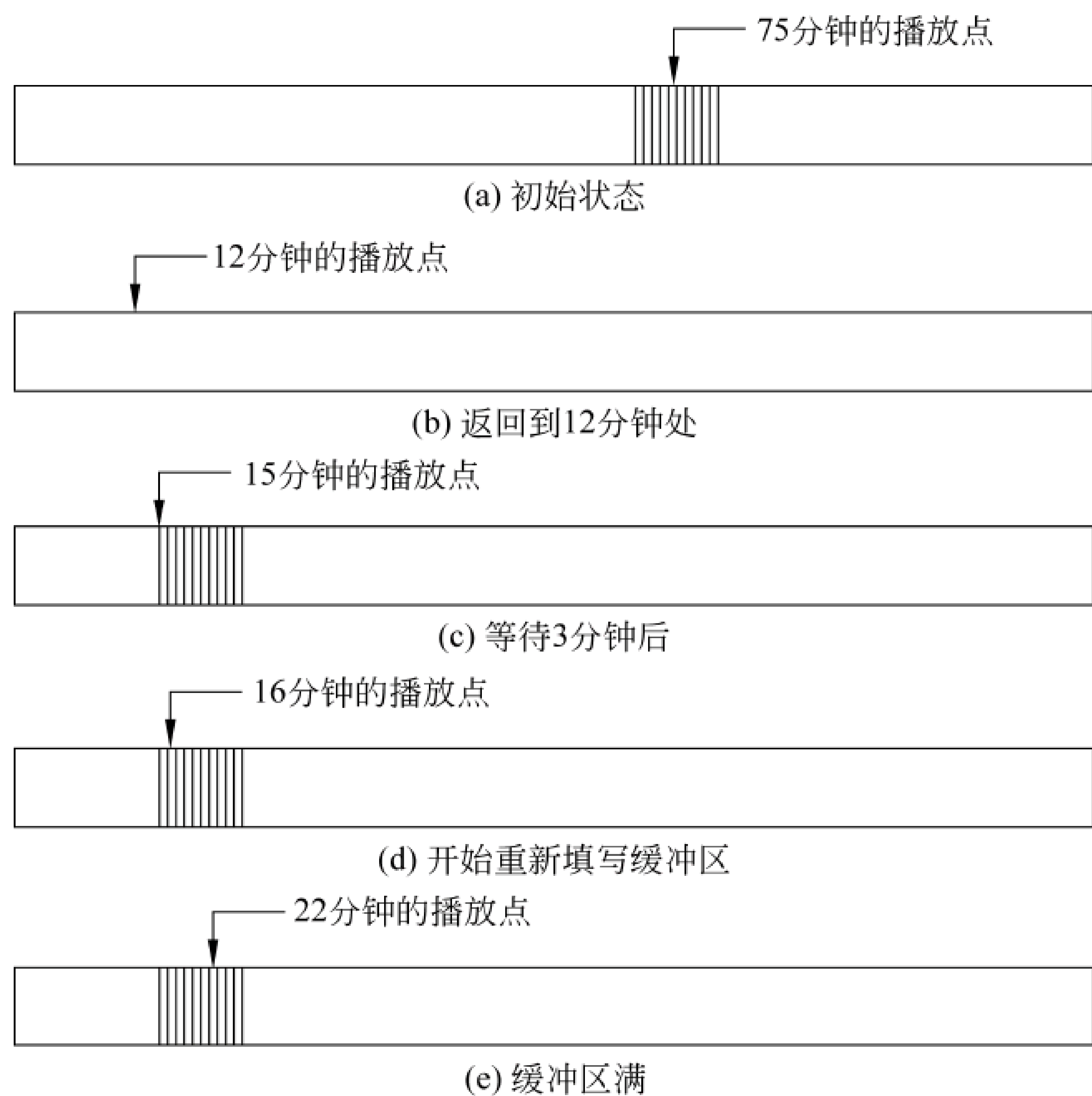


图9.16 缓冲区的使用情况

在增加了6分钟之后,缓冲区满了。播放点在22分钟。现在播放点不在缓冲区中间,如果必要的话可以调整。

9.6.5 文件在磁盘上的放置

多媒体文件很大,写一次,可读多次,一般是顺序访问,因此多媒体文件也称连续媒体(Continue Media,CM),播放它们必须满足严格的服务质量标准。这些要求导致多媒体操作系统的文件系统与传统操作系统使用的文件系统不同。

1. 单磁盘上的文件放置

对于多媒体播放最重要的要求是数据以请求的速度没有抖动地在网络上传输或在输出设备上输出。因此，在以帧为单位的传输中不希望有多种查找。消除在一个视频服务器上对内部文件查找的方法之一是采用邻接文件，一般情况下使用邻接文件不能很好地工作，但是在视频服务器上，事先必须加载电影，在它工作之后就不能改变，因而用邻接文件。

然而复杂的是视频、音频和文本都存在，如图 9.13 所示。即使视频、音频和文本文件都作为独立的邻接文件存在，如果需要的话，将有从视频到音频和从音频到文本的查找。这里介绍另一种存储方式，视频、音频和文本文件如图 9.17 所示交叉存放，但整个文件一直是邻接的。这里，帧 1 的各种音频索引和文本索引直接跟踪帧 1 的视频。依据音频和视频索引多少，以单磁盘上读操作方式读一帧的所有内容，仅传输必要的部分给用户，这种方式可能是最简单的。

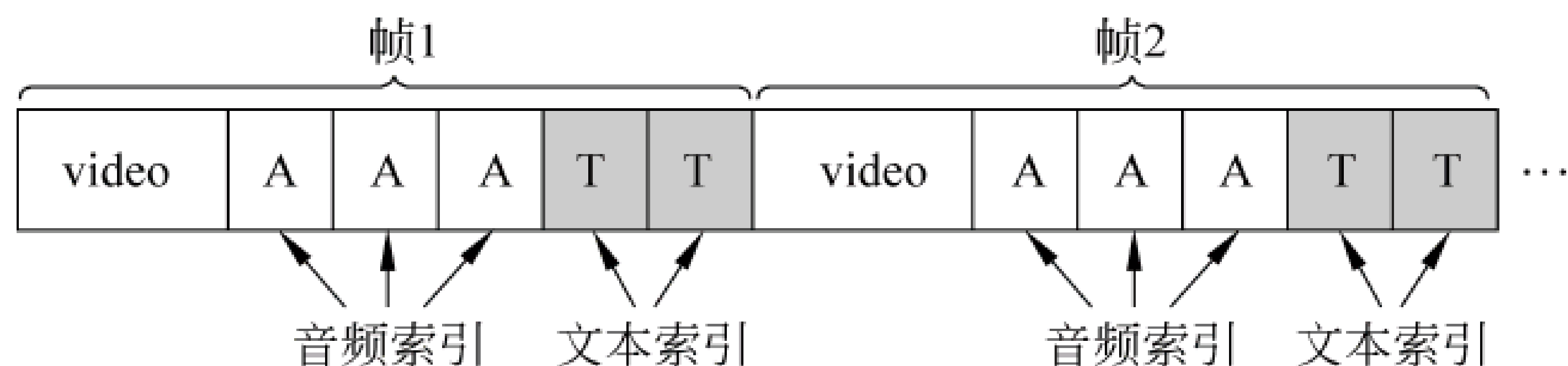


图 9.17　在每部电影的单一邻接文件中视频、音频和文本间隔存放

由于读了不希望的音频和文本内容，这种放置方式要求额外的磁盘 I/O 并且需要额外的内存缓冲区来存储它们。如果排除所有的查找（在单用户系统中），即使整部电影以一个邻接文件存在，为了保留帧在磁盘上的位置的索引，也需要任何额外的资源。在这种方式中随机访问是不可能的，如果不必要的话，失去它是允许的；同样由于没有附加的数据结构，快进和快退也是不可能的。

在具有多并行输出的视频服务器上，整部电影作为一个邻接文件的优点失去了，因为从一部影片中读一帧，在返回到该电影之前，不得不从磁盘上的许多其他影片中读一些帧。在一个影片被读也被写的系统中（例如使用和编辑视频产品的系统），使用巨大的邻接文件来操作是困难和不实用的。

2. 文件的放置策略

1）一部电影在磁盘上的放置

对于多媒体文件还有其他两种放置方式。

第一种放置方式是利用小磁盘块方式，如图 9.18(a)所示。在这种放置方式中，磁盘块的大小比帧平均大小小很多，即使 P 帧和 B 帧也是如此。对于 30 帧/秒、数据率 4Mb/s 的 MPEG-2，平均帧的大小是 16KB，所以块的大小为 1KB 或 2KB 比较好。其基本思想是有一个数据结构——帧索引，每部电影的索引记录每一帧的开始点。对于磁盘块是邻接的帧，每一帧包含视频、音频和文本索引。这种方法读第 K 帧包含通过帧索引检索找到第 K 帧的位置，然后以一次磁盘操作读整个帧，即使帧的大小不同（帧的大小需记录在帧索引里），帧索引也与 1KB 的磁盘块大小相等，一个 8 位的块能处理帧的大小可达 255KB，对于非压缩的 NTSC 帧是足够用的，即使具有许多音频索引。

第二种放置方式是利用大磁盘块（如 256KB）方式，在一块中放许多帧，如图 9.18(b)所示。需要一个索引，该索引是块索引而不是帧索引。事实上这一索引基本上同 i 节点

(inode)相同,可能需要增加信息告诉在一块中哪一帧是第一帧,这样使得快速定位给定的帧成为可能。

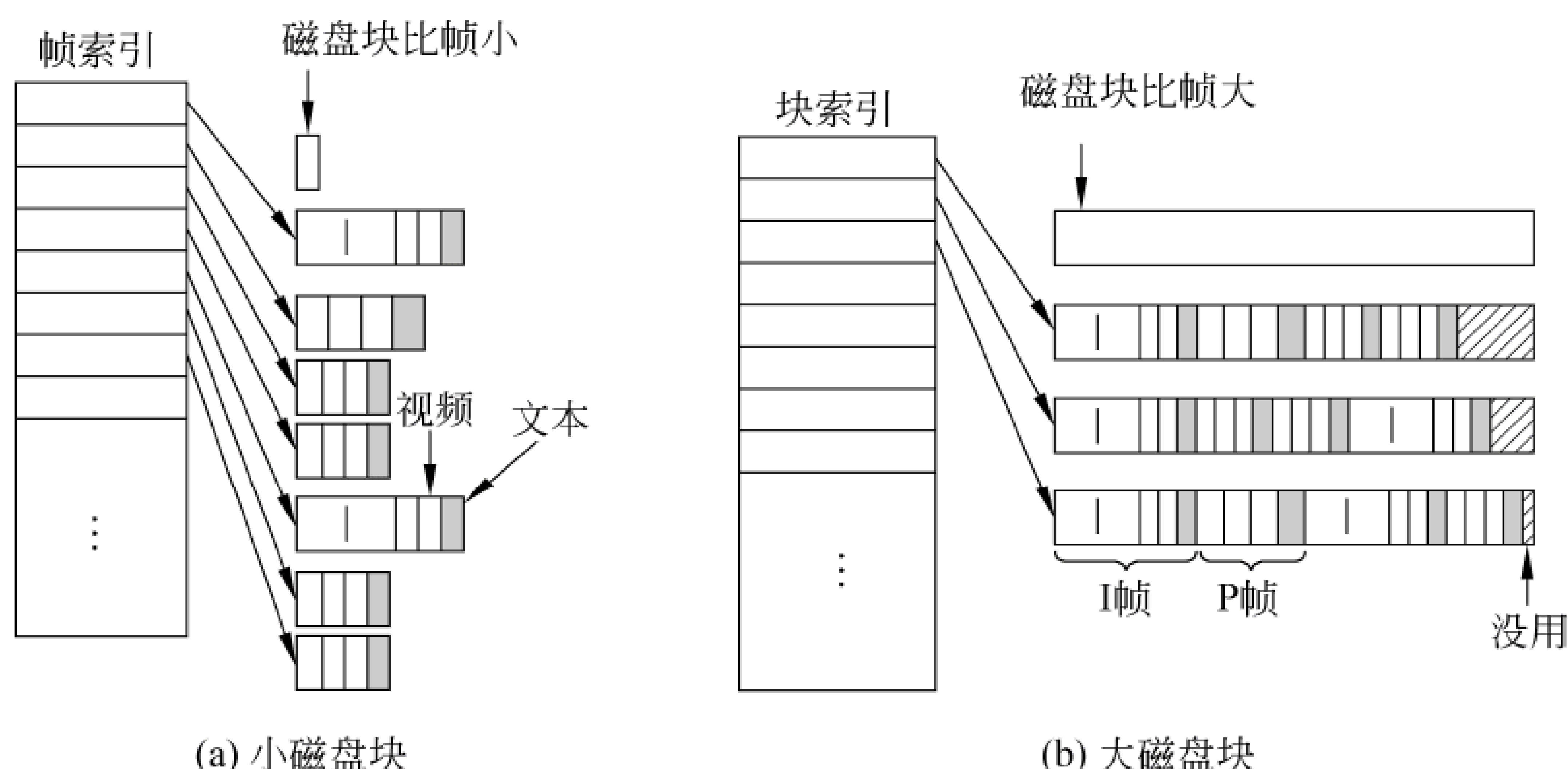


图 9.18 非链接形式的电影存储

这两种方法的不同之一是缓冲区。用小块的方法,每次读到一个确切的帧。因此,用一个简单双缓冲策略就可以实现:为当前播放的帧建立一个缓冲区,为下一播放帧建立另一个缓冲区。如果使用固定大小的缓冲区,每个缓冲区必须容纳最大的 I 帧。如果缓冲区的大小不同,在一个缓冲池中按帧的大小分配,那么帧的大小在帧读入之前就应知道,一个小的缓冲区可以被选作 P 帧或 B 帧使用。

用大块的方法需要一个复杂的策略,因为每个块包含多个帧,在每一块的结尾可能包含帧的碎片(依据前面所做的选择)。如果显示或传输帧要求块是邻接的,它们必须被复制,但复制是一种昂贵的操作,因此复制操作应尽量避免。如果邻接不被要求,那么跨越块的帧在网上传输或显示在设备上的数据量是两大块。

双缓冲区也可能用在大块方法中,但是用存放两个大块的内存做缓冲区必然浪费内存。减少内存浪费的一种方法是有一个循环传输的缓冲区,它比磁盘块(每帧)稍大。该磁盘块内容被送入网络和显示。当缓冲区的内容调整到某一阈值以下,一个新的大块从磁盘中读入,内容复制到传输缓冲区,大块缓冲区归还到公用池中,循环缓冲区的大小必须被选定,这样当达到阈值时,为另一磁盘块提供了空间。磁盘块的内容不能直接送到传输缓冲区,因为它必须有交换,在这里复制和内存使用之间进行折中选择。

这两种方法比较的另一个方面是磁盘效率。用大块方法可以用最高速度操作磁盘,这通常是受到关注的主要问题。把 P 帧和 B 帧作为一个分离的单元来读是低效的。另外,在多个驱动器上划分大块是可能的,而划分非独立块是不可行的。

图 9.18(a)所示的小块放置方式有时也称为固定时间长度,因为在索引当中的每一索引代表着相同的播放时间的值。相应地,图 9.18(b)所示的方式有时被称为固定数据长度,因为数据块的大小是相同的。

这两种文件放置方式的另一个不同点是如果帧的类型被存储在如图 9.18(a)所示的索引中,由于恰好显示 I 帧,因而完成一个快进处理是可能的。然而依赖 I 帧在流中出现的频率,当太快或太慢时,该频率可能被察觉。在任何情况下,具有图 9.18(b)所示的放置方式时快进是不可能的。事实上,为了找出所要的帧,顺序读文件要求进行大量的磁盘 I/O

操作。

2）多部电影文件在磁盘上的放置

至此，仅讨论了对一部电影的放置。事实上在视频服务器中存放很多部电影。如果它们随机地分布在磁盘上，当多部电影被不同用户同时观看时，从一部电影转到另一部电影时，磁头需要移动，浪费一定的时间。

通过观察可以得知：在磁盘上存放的电影中，一些比另一些流行。因而可以考虑利用流行性来安排电影的放置。磁头的移动时间可能减少。按流行性人们选择电影的行为符合 Zipf 定律，该定律是由美国的一名语言教授 George Zipf 发现的。该定律陈述的是：如果电影、书、网页或单词按它们的流行性进行排序，用户选择序列中的第 K 个条目的可能性是 C/K，这里 C 是常数。

这样选择前 3 部电影的百分比是 $C/1$、$C/2$ 和 $C/3$，相应地，通过所有项之和为 1 来计算 C 的值。换句话说，如果有 N 部电影，则

$$C/1+C/2+C/3+\cdots+C/N=1$$

根据这个公式，C 被计算出来。

对于存放在服务器上的电影，Zipf 定律说明选择最流行的电影的人数是选择第二部流行电影人数的两倍，是选择第三部电影的三倍，以此类推。尽管事实是：在开始分布很快下降，但它有个很大的尾巴。例如第 50 部电影的流行系数是 $C/50$，第 51 部电影的流行系数是 $C/51$，所以第 51 部的流行系数是第 50 部的 50/51，仅差大约 2%。越接近尾部的相邻电影之间的差异越小。一个结论是即使前 10 部电影满足了基本要求，但服务器还需存放大量电影。

了解不同电影的流行性对建立视频服务器上的效果模型和文件放置是大有帮助的。研究显示，最好的策略是特别简单并且分布独立。它被称为元件管道算法（organ-pipe algorithm，由 Grossman 和 Silverman 于 1973 年提出，该算法的名字的由来是：概率的直方图看起来像一串微微倾斜的元件）。它的基本思想是：把最流行的电影放在磁盘中间，第二和第三流行的电影分别放在它的两边；它们的外边分别放第四和第五流行的电影，以此类推，如图 9.19 所示。

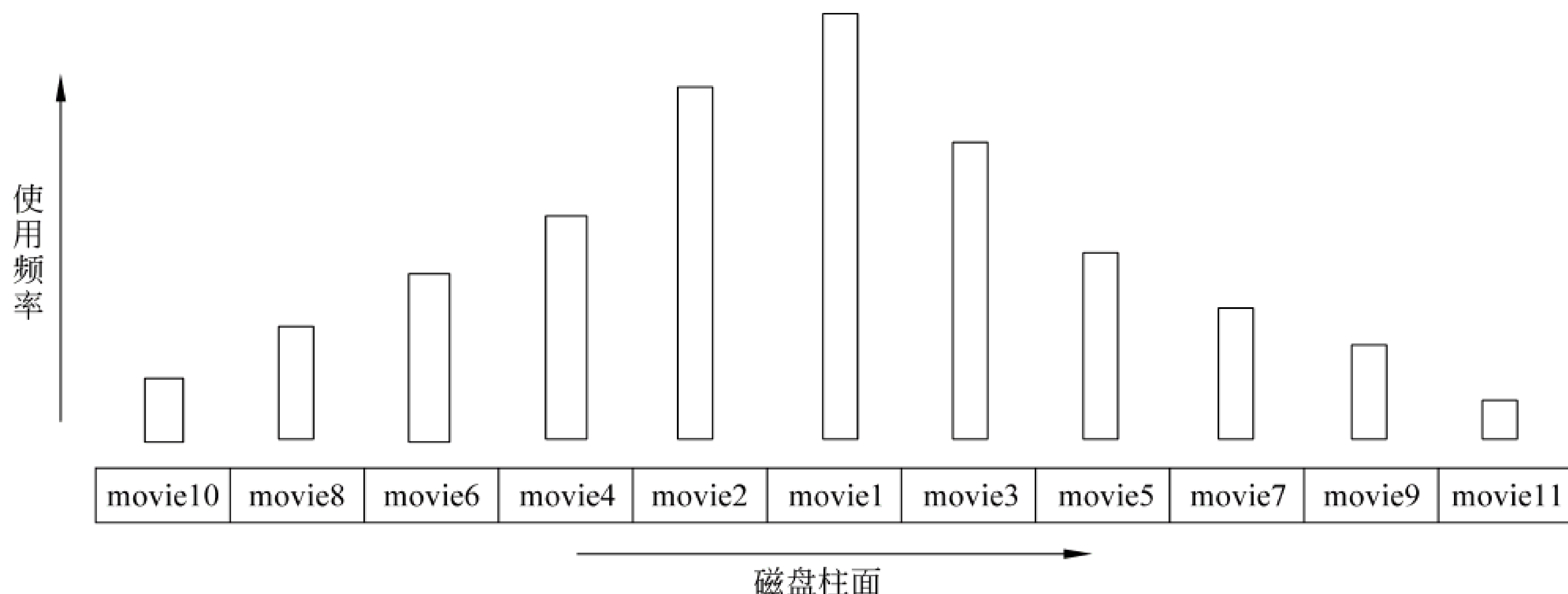


图 9.19　在服务器上的文件的元件管道分配

如果每部电影是如图 9.19 所示的邻接文件，则这种放置是最有效的，但如果电影被连

续地放在很窄范围的柱面上，也可用某些限定。

算法应做的是设法使磁头定位在磁盘中间，有100部电影按Zipf定律分布，头5部电影被选择的可能性之和为0.307，这意味着磁头位于头5部电影所在的柱面的概率为30%。

3. 准点播的文件放置

对于准点播，采取与上述文件放置方式不同的文件放置策略会更有效。记住同一部电影作为多交叉流播放。即使电影作为邻接文件被存储，对于每个流的查找也是需要的，Chen和Thaper于1997年提出了一个文件放置策略来消除几乎所有的查找。对于一部以30帧/秒、每隔5分钟开始一个新流(如图9.20所示)的策略的说明如图9.20所示。具有如图9.15所示例的参数，24个当前流对于2个小时的电影是必需的。

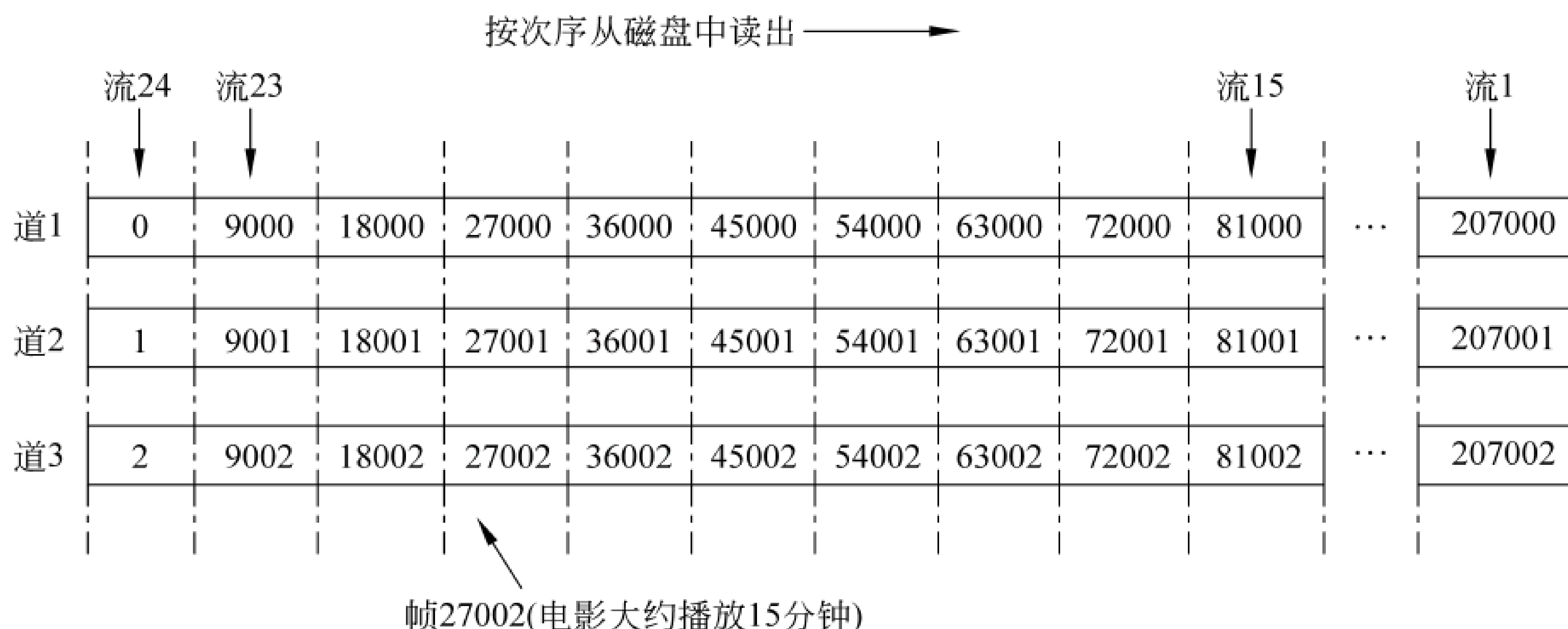


图9.20 为准点播优化帧的放置

在这种放置中，包含24个流的帧系列被连接起来，作为一个独立的记录写到磁盘上。它们也能一次性地读出来，考虑流24刚开始的时刻。它需要帧0，流23(前5分钟开始的)需要帧9000，流22需要帧18000，依此类推，流0将需要帧207000。连续地放这些帧的索引在磁盘上，以倒序次序带有对帧0的查找，视频服务器能够满足所有24个流。当然这些帧是有序的，当最后一个流被播放之后，磁头臂移动到道2准备重新播放它们。这样的安排不需要整个文件是邻接的，但对一定数量的流提供了好的效果。

一个简单的缓存策略是用双缓存区，一个缓存区用来播放24个流，另一个缓存区事先被装载。在当前的一个流结束时，在单一磁盘操作中两个缓存区交换，用来播放的一个缓存区恰好用来装载。

一个有趣的问题是缓存区要多大。很明显，它必须容纳24个帧，然而由于帧的大小是变化的，从整体看确定缓存区的大小是重要的，使缓存区大到能存24个I帧是过大，但是使缓存区大小能存平均大小的24帧是危险的。

幸运的是，对于任何给定的电影，在电影中最大的道数(如图9.15所示)事先是知道的，所以实际上缓存区的精确大小是可选择的。然而下面的情况可能刚好发生：在最大的道中有16个I帧，而下一个最大道仅有9个I帧。决定选择一个满足第二个最大道大小的缓存区可能是明智的。做出这一选择意味着截短最大的道。这样在一部电影中将拒绝某些流的帧。为了避免闪烁，前面的帧可能重播，但没人能注意到。

进一步利用该方法，如果第三个最大道仅有4个I帧，使用能存4个I帧和20个P帧的

缓冲区是值得的。在一部电影中对某些流二次引进两个重复的帧是可以接受的,以此类推,缓冲区的大小对 99%的帧够用即可。显然在提供作为缓冲区的内存的使用和电影播放的质量之间存在折中。同步的流存在得越多,统计的结果越好,帧系列将更一致。

4. 多个磁盘上的文件放置

为取得高效,视频服务器通常装有许多磁盘,它们能并行工作。有时使用 RAID(磁盘冗余阵列),但不常用,因为 RAID 提供的是以效率为代价的高可靠性。通常视频服务器希望高效率,不太关心纠正瞬间的错误。如果一次有太多的磁盘被处理,RAID 控制器可能成为瓶颈。

最简单而普遍的配置是大量的磁盘,有时被作为磁盘器庄(disk farm)。在同步方式中磁盘不能回转,不能包含任何奇偶效验位,而 RAID 能。一种可能的配置是电影 A 放在磁盘 1 上,电影 B 放在磁盘 2 上,以此类推,如图 9.21(a)所示。事实上可以在每张磁盘分别放不同的电影。这种放置实现起来简单,但有直接失效的特性:如果一张磁盘有问题,在它上面所存的电影都将失效。这种方法的另一缺点是装载可能是不平衡的。如果一些磁盘存放流行的电影,另一些磁盘存放非流行的电影,系统将不能被有效利用。当然,一旦知道电影的使用频率,通过手工转载使它们达到平衡是可能的。

第二种可能的放置是把每部电影进行划分,分布在多磁盘上,在图 9.21(b)所示的例子中有 4 部电影。

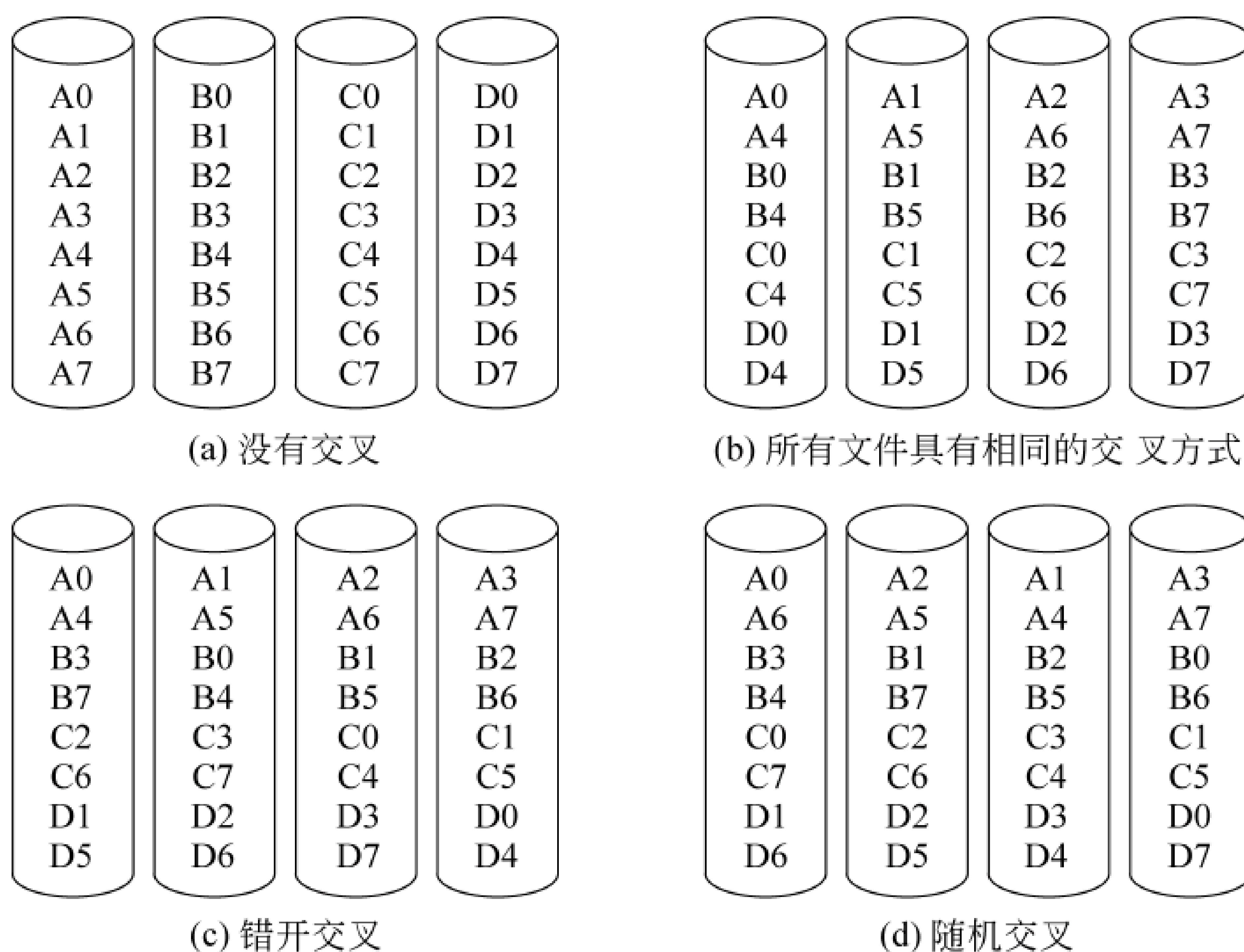


图 9.21 在多个磁盘上组织多媒体文件的 4 种方法

这种划分方式可能的缺点是:因为所有的电影都从磁盘 1 开始,跨越磁盘的装载可能不平衡。扩展装载的一种较好的方法是错开开始的磁盘,如图 9.21(c)所示。另一种试图平衡装载的方法是对于每个文件用随机划分的方法,如图 9.21(d)所示。

至此,假设所有帧的大小相同,对于 MPEG-2 格式的电影,这种假设是错误的:I 帧比 P 帧大。有两种方法可用来处理这种情况:按帧划分和按块划分。当按帧划分时,电影 A 的第一帧放在磁盘 1 上,其所占磁盘块的大小与帧的大小相同,在同一磁盘上的块是邻接单

元,下一帧放在磁盘 2 上,以此类推。电影 B 以相似方法划分,可以在同一磁盘上开始,也可以在下一个磁盘上开始(如果是交错的话)或随机的一个盘开始。由于一次读一帧,这种划分方式不能提高对任意给定电影的读取进度。然而它在整个磁盘上的装载比图 9.21(a)要好,如图 9.21(a)所示,如果今天晚上有很多人看电影 A,没有人看电影 C,则情况比较坏。从整体上说,在所有磁盘装载使得磁盘带宽的总和可用,这样增加了可服务的用户数量。

划分的另一种方法是按块划分,对每部影片划分为固定大小的单元,按次序或随机地写到多个磁盘中的每个磁盘上。每块包含一个或多个帧或有碎片,现在在同一时刻对于同一部电影的多块发出请求,每一请求要求读数据到不同的内存缓冲区中,但按这种方法完成所有请求,包含许多帧的电影的邻接的块被分配在邻接内存中,这样请求能够并行处理。当最后请求被响应时,请求进程发出一个任务完成信号,然后传输数据给用户。

传输一定数量的帧以后,缓冲区中存有最后几帧时,更多的请求发出,把帧预先加载到其他缓冲区中。这种方法为了使磁盘忙而使用了大量内存作为缓冲区,在一个具有 1000 个活动用户和 1MB 缓冲区(例如在 4 个磁盘上每个都以 256KB 为一块),需要准备 1GB 大小的缓冲区。在 1000 个用户的服务器上这个数量是微不足道的,不会有问题。

关于划分的一个决定性的问题是在多少个磁盘上进行划分。一种情况,每个电影划分在所有磁盘上。例如,对于 2GB 的电影有 1000 个磁盘,则 2MB 大小的块被写到每个磁盘上,所以没有一个电影读一个磁盘二次,其缺点是一个磁盘的失效将影响所有的电影。另一种情况是磁盘被分成小组(如图 9.21 所示),每部电影被限定存储在一组磁盘上。前者称为粗条纹划分,对于在所有磁盘上平衡负载有利。后者称为细条纹划分,它得承受热点(流行的组),但是一个磁盘的失效仅涉及在这组磁盘上的电影。

9.6.6 缓存

传统的 LRU 文件缓存对于多媒体文件来说不能很好地工作,因为电影的访问方式与文本文件的访问方式不同。传统的 LRU 缓存的思想是一个块使用之后,它将保存在缓存中,很快地再次被需要,例如当编辑一个文件时写有文件的块通常被不断地使用,直到编辑任务结束。换句话说,在一个短的时间内一个块被重复使用的可能性高时,为了避免再次访问磁盘,把文件保存在缓存中是值得的。

而对于多媒体文件来说,通常的访问方式是电影从头到尾被顺序访问。块通常不用第二次,除非用户回退电影再看某些情节。相应的一般的缓存技术没有效果。然而对于多媒体缓冲也能提供帮助,仅仅是使用方式不同。

1. 块缓存

虽然希望可能被快速的重复使用而保留的块是不可能的,但为了使缓存的多媒体再被利用,需要研究多媒体系统的预测能力。假设两个用户看同一部电影,一个用户开始 2 秒钟后另一个用户才开始。第一个用户得到并看完了所有块后,第 2 个用户在 2 秒钟后看相同的块是可能的。系统很容易保留这样的线索:哪部电影仅有一名观众,哪部电影有两名或多名观众,在相近的时间内被观看。

这样用任何时刻以一部电影的名义读的一块,在短时间内将再次被使用。这种情况需要缓存它,依据需要缓存的大小、内存的紧张程度,不采用在缓存中保留所有的磁盘块的方法,而是采用当缓冲满时淘汰最近很少用的块的方法;而用另外策略,在第一个观众的 ΔT

时间后有第二个观众的每部电影作为能缓存的被标记,它的所有块被放入缓存直到第二个(也可能是第三个)观众使用它。而对于其他电影则不缓存。

这种思想可进一步发展。在一些情况下,合并两个流是可行的。假设两个用户看同一部电影,但时间上相差 10s 以内。该块在缓存保留 10s 是可能的,但浪费内存。一种十分隐蔽的替代方法是使得两部电影合并,这通过变更两部电影的帧率来实现,如图 9.22 所示。

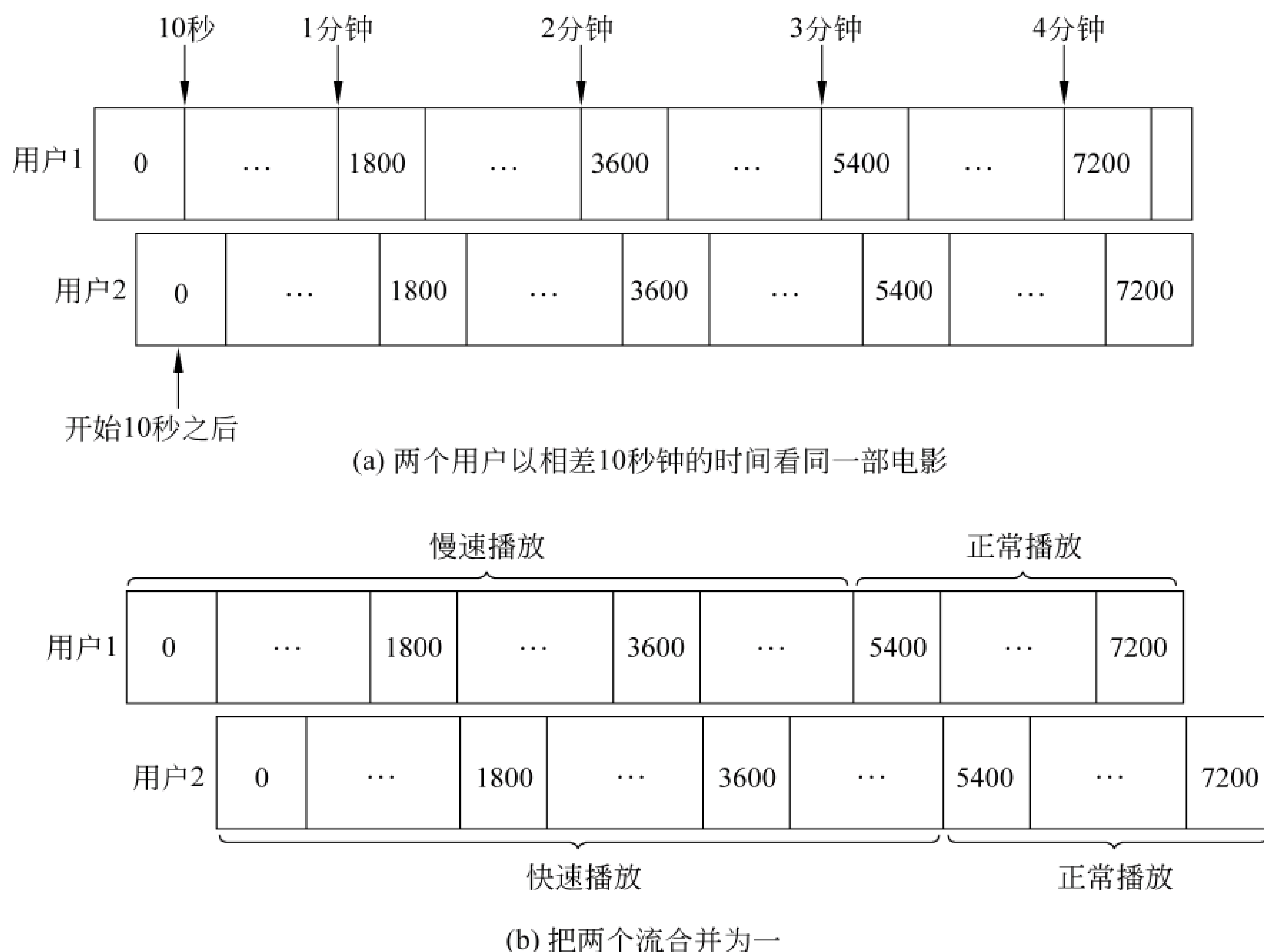


图 9.22 把两部电影合

2. 文件缓存

缓存在多媒体文件中也有其他使用方式。由于大部分电影较大(2GB),视频服务器不能把所有的电影存储在磁盘上,而把它们存储在 DVD 或磁带上,当需要一部电影时把它复制到磁盘上,但有一个具体的启动时间来确定电影的存放位置并复制到磁盘上。因而大部分视频服务器对请求比较多的电影拥有一个磁盘缓存,流行的电影被完整地存在磁盘上。

使用缓存的另一种方法是在磁盘上保留每部电影的头几分钟的部分。当一部电影被请求时,能够立即从磁盘的文件开始处播放。接着把电影从 DVD 或磁带复制到磁盘上,通过在磁盘上保存足够的部分,在电影的下一部分请求到达之前被取得的可能性是很高的。当整部电影都播放完,其内容在其他请求到达之前都将在磁盘上,在以后有更多的请求情况下把它放在缓存中,存在磁盘上。如果长时间没有用户请求,该部电影将从缓存中移去,留出空间存储更流行的电影。

9.6.7 多媒体磁盘调度

多媒体按不同的要求放在磁盘上,而与编辑器和文字处理器等传统的面向文本的应用有所不同。特别是多媒体要求播放的高数据率和实时数据播放。它们当中的任何一个都不

是普通的规定。更进一步，在拥有一个多媒体服务器的情况下，该服务器同时处理几千个客户有巨大压力。这些请求影响整个系统。

1. 静态磁盘调度

虽然多媒体在整个系统中具有实时和高数据率的要求，但是它也有一个特性使其处理起来比传统的系统容易——预测性。传统的操作系统以无预测的方式处理磁盘块的请求。磁盘子系统只需读取每一个打开的文件的第一块；另外它能做的就是等待请求到来，按要求处理它们。而多媒体系统则不同，每一个活动流把定义好的负载加载到系统上，它有很高的预测性。对 NTFC 播放形式，每 33.3ms 每个用户想得到文件的下一帧，系统有 33.3ms 得到所有的帧（系统至少缓存一帧，所以 $k+1$ 帧的获取在播放 k 帧的同时进行）。

这种有预测的加载（为多媒体操作而编制的算法）用在磁盘调度上。下面只考虑单磁盘的情况（这种思想也被用在多磁盘上）。在这个例子中，假设有 10 个用户，他们看不同的电影，更进一步假设有相同的分辨率、帧率和其他特性。

依据系统的剩余资源，计算机可以有 10 个进程，每个视频流为一个进程，或是一个具有 10 个线程的进程，或具有能用时间片轮转方式处理 10 个流的一个线程的进程。细节是不重要的，重要的是时间被分成片，这里一个时间片是处理一帧所需的时间（对 NTSC 方式是 33.3ms，对 PAL 是 40ms）。在每一时间片的开始按每一个用户的行为产生一个磁盘请求，如图 9.23 所示。

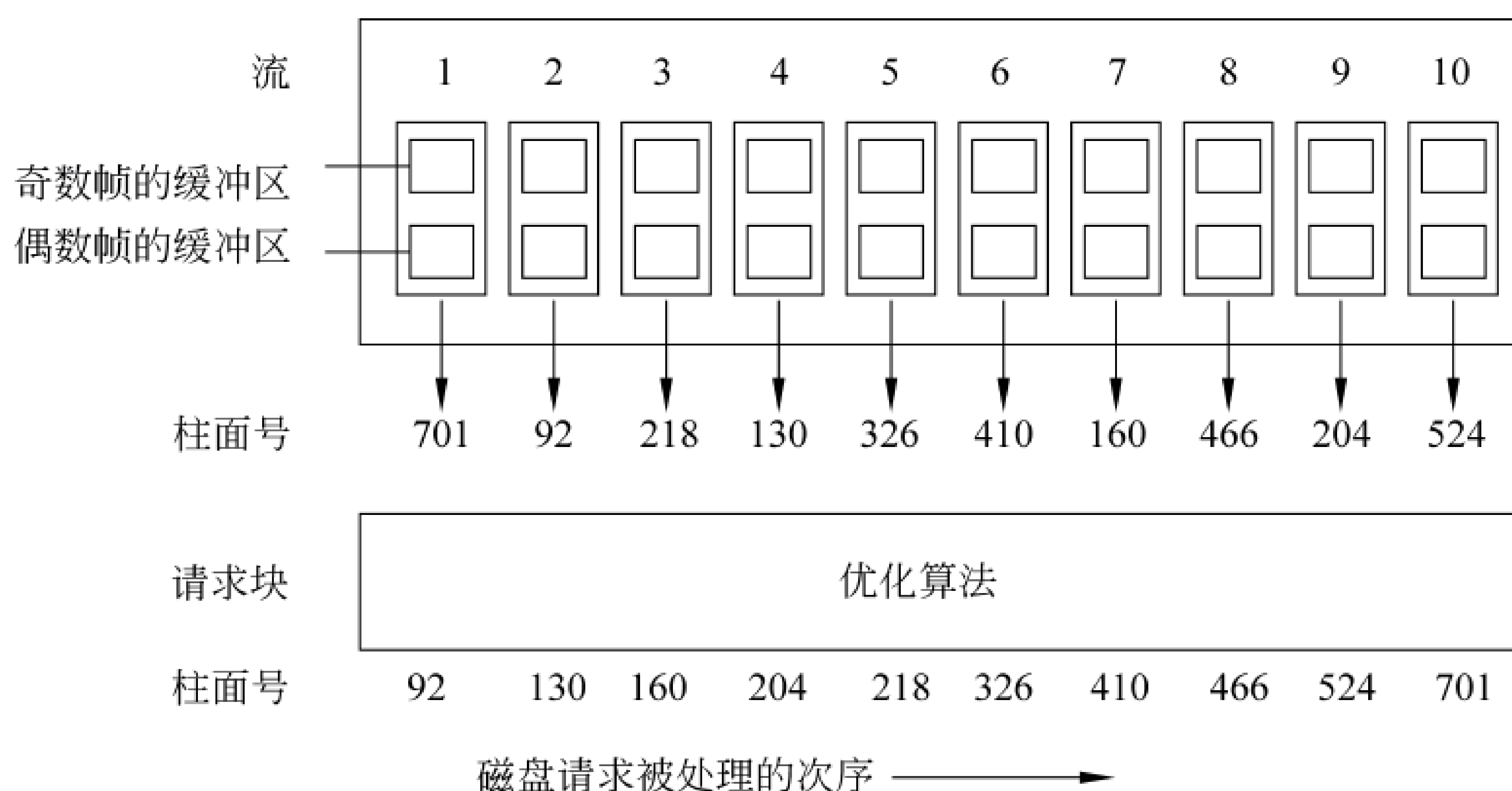


图 9.23 在一个时间片内每部电影要求一帧

在时间片开始处，所有请求都达到之后，磁盘知道在时间片内应该做什么，它也知道在所有的请求处理完和下一个时间片开始之前没有其他请求。因此按优化方法对请求排序，图 9.23 显示了按柱面顺序排列的请求。

按这种方式排序看起来似乎没有意义，因为只要磁盘满足时限，无论有 1ms 的剩余还是有 10ms 的剩余是无关紧要的。然而，这个看法是错误的。通过这种方式的优化查找，处理每一个请求的平均时间被缩短，这就意味着平均来说在单位时间里磁盘处理更多的流。换句话说，这种优化磁盘请求的方法能够增加服务器传输的电影数量。在时间片末尾的剩余时间，可以被用来对非实时请求提供服务（如果存在的话）。

如果一个服务器有太多的流，一旦要求从磁盘中远的部分获得帧，就可能超过了时限。

只要被贻误的时限特别少,为了处理更多的流它们被默许立即返回。注意,重要的是获得的流的数量。每个流有两个或多个用户不影响磁盘的执行效果和调度。

为了使流平滑地到达用户,在服务器中双缓冲区是必要的。在时间片1内一个缓冲区被使用,每一个流占有一个缓冲区。当时间片结束时,输出进程或线程被唤醒,被告知传输帧1。在同一时刻对每部电影帧2的新请求到达(对于每个电影可能有一个磁盘线程和输出线程),当第一个缓冲区一直忙时,用第二个缓冲区满足这些请求。当时间片3开始时第一个缓冲区空闲,为了获得帧3而被使用。

假设每帧有1个时间片(这一限制不是非常必要的)。如果每帧有2个时间片,则减少了所需缓冲区空间大小。对于许多磁盘操作花费增加一倍。类似地在每一个时间片从磁盘上获得两个帧(假设帧成对在磁盘上存储)。这种设计磁盘操作的次数减少一半,所需要的缓冲区空间增加一倍,依据相对可用性效率,内存对磁盘I/O的比优化策略可计算并被使用。

2. 动态磁盘调度

在上面的例子中做了这样的假设:所有的流都有相同的分辨率、帧率和其他特性。现在调整这一假设,不同的电影有不同的数据率,因此每33.3ms是一个时间片,每个流获得一帧是不可能的;另外,对磁盘的请求到达的时间也是随机的。

每一个读请求说明读哪一块、何时需要该块(即它的时限)。为简化起见,假设对每一请求的实际服务时间是相同的(实际上不一定)。在该方法中从得到请求到请求处理结束的时间减去这个固定时间(服务时间),时限始终得到满足。这样使得模型简化,因为磁盘调度关心的是请求的时限。

当系统开始时,没有对磁盘的请求。当第一个请求到达时,它立即得到服务。当第一个请求被处理时,其他请求可能到达,所以当第一个请求结束时,磁盘驱动器可以从这些请求中选择一个进行。一些请求被选中并开始处理。当请求处理结束时,又可能有一系列的请求:第一次没有选中的请求,第二个请求处理时新到达的请求。通常无论什么时候处理一个请求,驱动器伴有其他请求,可以从中进行选择。问题是用什么算法选择下一个请求进行处理。

两个因素在选择下一个磁盘请求时起作用:时限和柱面。从效率的观点看,保持按柱面排序的请求;利用电梯调度算法使总的查找时间最少,但访问外层柱面的请求可能超出的时限。从实时处理的观点看,按时限排序请求,最早时限最早调度,则超出时限的机会最少,但这种方法增加了总的查找时间。

Scan-EDF算法(由Reddy和Hyllie于1992提出)对这些因素进行了综合。这个算法的基本思想是:把时限相近的请求收集在一组中,按柱面次序处理它们。例如,考虑图9.24在$t=700$的状态,磁盘驱动器知道它同时拥有10个不同时限、不同柱面的请求。作为一个例子,它决定把4个具有最早时限的请求作为一组,按柱面号排序,按柱面次序利用电梯调度算法进行调度。调度次序是110、330、676、680。只要在请求的时限之前完成,请求就能被安全地重排使查找所需的时间最少。

当不同的流有不同的数据率时,一个新用户出现会产生严重的问题:该用户是否被接受。如果接受这个用户,可能频繁地引起其他流贻误时限,则答案是可能不接受。有两种方法用于决定是否接受新用户。一种方法是假设每一用户需要一定数量的资源(按平均计

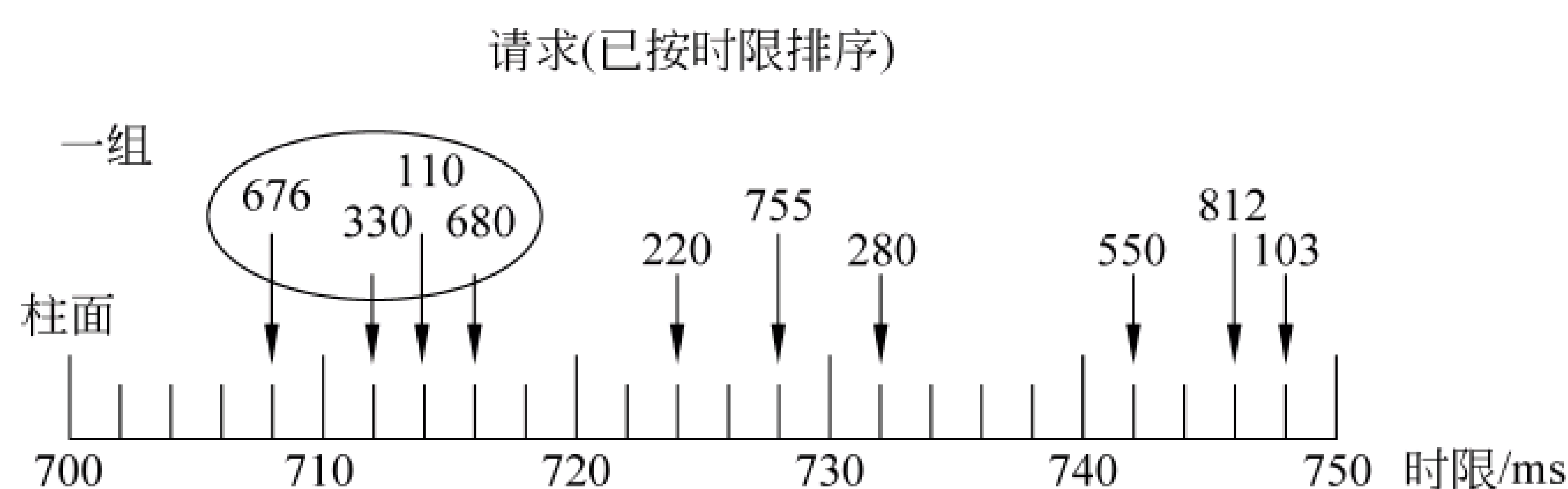


图 9.24 用时限和柱面号进行调度的 Scan-EDF 算法

算)，例如带宽、内存缓冲区和 CPU 时间等。如果对于一般用户，每一部分剩余资源都是足够的，新用户将被接受。

另一种算法比较细致，它对新用户想看的特定电影进行查找，对于该部电影检查电影的数据率(预先计算的)，是黑白还是彩色，是动画片还是一般电影，甚至爱情片与战争片都不相同。爱情片带有长镜头慢慢移动，慢慢消失；与这些形成对比，战争片有许多迅速切换的快动作，因而有许多 I 帧和大量的 P 帧。如果服务器对新用户想看的特殊电影有足够的容量，则接受该用户，否则拒绝该用户。

9.7 本章小结

本章首先介绍了安全和安全操作系统。威胁、入侵和意外数据丢失是造成系统不安全的最重要的 3 个方面。常采用密码术对传输的数据进行加密，以保证数据的安全。加密方法主要有两种：私有密钥和公开密钥。数字签名是目前一种常用的加密方法，它是把私有密钥和公开密钥结合的一种方法。保证非法用户不进入系统的措施是用户认证，即验证用户的身份，主要的验证方法有使用口令认证、使用物理对象的认证、使用生物技术的认证和计数测量。对系统的进攻有来自系统内的，也有来自系统外的。操作系统常用的保护机制是按保护域存取控制，保护域的管理方法有存取控制矩阵、访问控制列表和权能列表。目前被公认的安全模型主要有以下几种状态机模型、信息流模型、无干扰模型、不可推断模型和完整性模型。实际上，在操作系统安全中常涉及的安全模型主要有状态机模型和信息流模型。安全操作系统的开发方法有两种：采用从头开始建立一个完整的安全操作系统和基于非安全操作系统开发安全操作系统的方法。安全操作系统的开发过程分为 3 个阶段：建立一个安全模型、安全机制的设计与实现及安全操作系统的可信度认证。一般评测操作系统安全性的方法有 3 种：形式化验证、非形式化确认及入侵分析。这些方法可以独立使用，也可以将它们综合起来评估操作系统的安全性。操作系统的安全评测是目前国际上的研究热点，我国也在这方面做了许多工作，并取得了一定的成绩。

随后介绍了并行计算机系统的分类，按照 Flynn 的方法，目前有两种类型的并行计算机系统：SIMD 型和 MIMD 型。多处理器操作系统一般为一个分时系统，它最主要的特点是系统中存在单一的一个运行队列，系统内所有就绪进程在此排队。根据各处理器可以执行操作系统的情况，多处理器操作系统一般可以分成 3 类：主从式、浮动式和对称式。在多线程系统的多处理器调度中采取的方法和策略主要有：负载共享调度、负载绑定调度、、组调度和多级动态调度。

在简单介绍了集群系统之后，较详细地介绍了并行操作系统、网络操作系统和分布式操作系统的基本概念、构成环境、特点和主要功能。相对来讲，网络操作系统和并行处理中的多处理器操作系统是较为成熟的操作系统，而分布式操作系统尚处于研究发展阶段，还有许多关键技术及实现方法有待于进一步研究和探讨。目前，计算机网络的应用越来越普及，应用范围越来越广，所完成的功能越来越强，对于网络操作系统的要求也越来越高，而且网络技术本身也在快速发展变化，如主动式网络和无线网等，所以网络操作系统也处于不断发展的过程中。随着微电子技术和通信技术的发展以及相关新技术的出现，并行处理体系结构将逐渐得到发展，并行操作系统也将随之发生变化，将具有更强的并行任务处理能力；对于由多台计算机组成的分布式并行处理系统（如集群系统和网格计算环境等），进程迁移、负载平衡、机间通信、资源发现及管理将是该类操作系统所要解决的重要问题。

多媒体技术目前应用得非常广泛。由于多媒体文件大并具有实时播放要求，针对文本设计的操作系统对多媒体而言不是最优的。多媒体文件包括多个索引，通常有一个视频，至少有一个音频，有时还有字幕索引，在播放时，这些必须是同步的。视频压缩有内部压缩（JPEG）和外部压缩（MPEG）方式。后者有 P 帧，而前者没有 P 帧；B 帧既是前者的基础也是后者的基础。为了满足实现多媒体需要的实时调度，通常使用两个方法：频率单调调度算法和时限调度算法。多媒体文件系统通常采用推模式而不是拉模式。在多媒体系统中采用准点播，用户只能看指定时刻播放的电影。文件采用邻接或非邻接方式存储。在后一种情况中，单元的长度是变化的（一块为一帧）或固定的（一块包含几个帧）。在实际使用中一般采用折中方案。文件在磁盘上的放置方式根据 Zipf 定律安排，因而当存放多个文件时有时利用元件管道算法。把文件进行划分（有粗条纹划分和细条纹划分）再放在磁盘上是普遍采用的一种方法。改进播放效果的一种方法是采用块和文件缓存策略。多媒体的磁盘调度分为静态调度和动态调度两种方式。

习　题

1. “安全”和“保护”有何联系和区别？
2. 哪 3 个方面是影响安全的最重要的方面？
3. 针对计算机安全的 3 个目标的 3 种威胁是什么？
4. 什么是入侵者？常见的入侵者是哪几类人？
5. 引起数据丢失的主要原因有哪些？
6. 保证安全的基本措施有哪些？
7. 什么是密码术？加密算法由几部分组成？常见的加密算法有哪几种？
8. 什么是数字签名？它有何作用？
9. 用户认证的含义是什么？简述常见的用户认证方法。
10. 对计算机系统的进攻主要来自哪两方面？各方面中的主要进攻方法有哪些？
11. 在保护机制中“域”的含义是什么？主要的保护机制有哪些？
12. 访问控制列表和权能列表有何区别和联系？使用权能列表有何优点？
13. 在计算机系统中，软件安全是如何划分的？
14. Linux 系统采取了哪些安全措施？

15. 什么是安全操作系统？它经历了哪几个阶段？简述各阶段的主要特点。

16. 安全操作系统的模型有哪些特点和作用？分为几类？各类对开发安全操作系统有何影响？公认的模型有哪几种？

17. 安全操作系统开发方法有几种？基于非安全操作系统的开发方法中又分为哪几种？

18. 简述安全操作系统的开发步骤。

19. 操作系统安全评测基础是什么？评测方法有哪几种？试比较各种评测标准的异同。

20. 理解并说明 Flynn 对计算机系统的分类方法。

21. 并行计算机系统有几种实现方式？各种方式通过什么机制来实现并行操作？

22. 理解并说明 SMP 系统和 MPP 系统的异同和特点。

23. 对称多处理器系统有几种类型？具体为何？

24. 对称多处理器系统所完成的主要功能是什么？

25. 说明对称多处理器系统中的多处理器调度算法。

26. 说明计算机网络软件的种类和主要功能。

27. 说明网络操作系统的种类及具体特点。

28. 说明网络操作系统的主要功能。

29. 说明网络管理的主要功能。

30. 什么叫分布式计算系统？

31. 说明分布式操作系统的特点。

32. 说明分布式系统和计算机网络间的区别。

33. 什么是客户/服务器计算模型？

34. 分布式操作系统的特点是什么？

35. 理解并说明分布式系统的消息传递、远程过程调用等通信方法的实现原理和过程。

36. 理解并说明分布式系统中进程同步/互斥的复杂性。

37. 请描述分布式系统中 Lamport 算法的具体步骤。

38. 请描述分布式系统中进程互斥的令牌环算法。

39. 说明分布式操作系统资源管理的主要功能和方式。

40. 理解说明分布式操作系统中进程迁移的主要目的。

41. 什么是集群系统？集群系统中节点的连接方式是什么？

42. 解释什么是多媒体。多媒体技术主要应用在哪些领域？

43. 实现多媒体点播需要一种什么样的基础结构？

44. 多媒体的关键特性有哪些？

45. 多媒体文件主要包括哪几种类型的文件？为什么要对多媒体文件进行压缩？

46. 数据压缩的主要方法有哪几类？常用的数据压缩编码方法有哪几种？

47. 静态数据压缩采用何种标准？该标准采用何种压缩算法？

48. 运动数据压缩采用何种标准？在该标准中图像是怎样分类的？

49. 在多媒体处理中采用哪几种处理器调度算法？各适用于什么场合？

50. 在多媒体处理过程中的实时行为有何特殊要求？

51. 在多媒体处理过程中如何实现“快进”和“快退”?
52. 准点播的含义是什么? 如何实现准点播?
53. 多媒体文件在磁盘上是如何放置的? 各种放置方式有何优缺点?
54. 简述 Zipf 定律的主要内容。利用该定律如何改进多媒体文件的放置以提高效率?
55. 多媒体处理中的缓冲有哪两种含义? 为什么要进行块缓冲和文件缓冲?
56. 在多媒体处理中磁盘调度有何特点?
57. 在文件服务器上为什么设置双缓冲?
58. 在多媒体处理中如何判定是否接纳一个用户为新用户?

第 10 章　操作系统实验

10.1　编程接口实验

1. 实验题目

编程接口实验

2. 实验目的

本实验通过使用系统调用编制程序，加深对操作系统提供的编程接口的理解。

3. 实验预备内容

复习 C 语言的常用库函数的功能和使用方法和汇编语言的常用系统调用。

4. 实验内容

利用 C 语言 5 个常用库函数和汇编语言的常用 5 个系统调用分别编写一段程序，使用的函数和系统调用不限，完成的功能也不限。

5. 提示

样例　用 gets、strcmp、strcpy、strlen 和 printf 共 5 个库函数完成 3 个字符串的比较，并求最大的一个字符串长度，输出最大个字符串及其长度。

用 C 语言实现的程序如下：

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str[20];
    char s[3][20];
    int i,len;
    for (i=0;i<3;i++)
        gets(s[i]);
    if (strcmp (s[0],s[1])>0)
        strcpy(str,s[0]);
    else strcpy(str,s[1]);
    if (strcmp (s[2],str)>0)
        strcpy(str,s[2]);
    len=strlen(str);
    printf("\n the largest string is %s\n the length is %d !\n",str,len);
    return 0;
}
```

10.2　进程管理(创建、执行和终止)实验

1. 实验题目

进程的创建、执行和终止实验。

2. 实验目的

本实验的目的是通过使用 Linux 的系统调用 fork()、exec()、exit()编写一个程序，加深理解进程的创建、执行和终止等内容。

3. 实验预备内容

(1) 理论中的进程和程序的概念，明确进程和程序的区别。

(2) 阅读与 Linux 系统中进程管理有关的函数内容，了解函数 fork()、exec()和 exit()等的功能和使用方式。

4. 实验内容

(1) 利用 fork()函数创建两个子进程。让系统中的 3 个进程分别输出一个不同的字符。观察并记录屏幕上显示的结果，分析原因。

(2) 利用 fork()和 execlp()函数实现一个 shell 的基本功能，如图 10.1 所示。用户输入命令后，按下列步骤执行用户命令：

① 利用 fork()函数创建一个子进程。

② 利用 execlp()函数启动命令程序。

③ 利用 wait()函数使父进程和子进程同步。

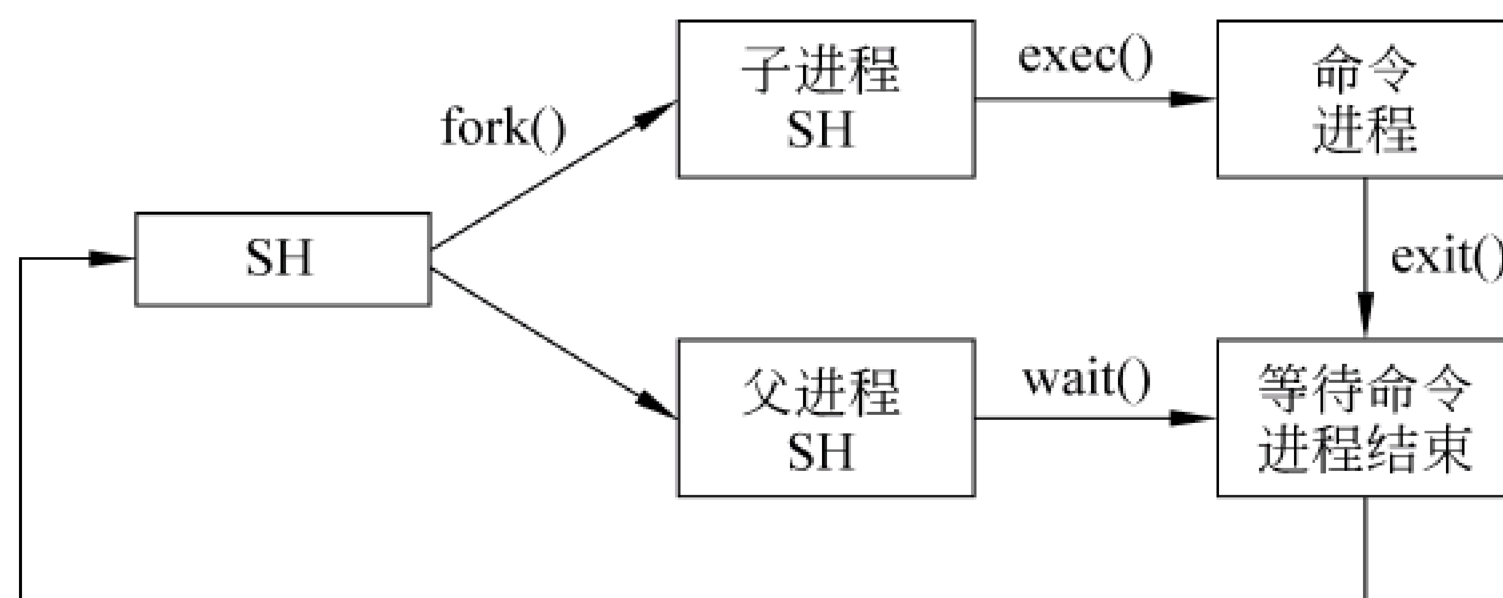


图 10.1　shell 执行过程

5. 提示

样例 1　利用 fork()和 execlp()函数实现一个 shell 的基本功能的 C 语言的程序如下：

```
#include <stdio.h>
int main()
{
    char command[32];
    char * prompt="$ ";
    while (printf("%s",prompt),gets(command) != NULL)
    {
        if (fork()==0)
            execlp(command,command,(char * ));
        else
            wait(0);
```

```
    }
    return 0;
}
```

样例 2 利用 fork()函数创建子进程,利用 exit()函数终止被创建的子进程,用 C 语言实现的程序如下:

```
#include <stdio.h>
main()
{
    int p1,a,b,sum;
    char * prompt="$ ";
    while ((p1=fork())==-1);
        if (fork()==0)
        {
            printf("The child process is running!");
            scanf("input two integers: %d,%d", &a,&b);
            sum=a+b;
            printf("sum is %d",sum);
            exit(0);
        }
        else
            printf("The father process is running!");
    wait(0);
    return 0;
}
```

注:该实验程序在 Linux 环境中执行。

10.3 作业(进程)调度实验

1. 实验题目

作业(进程)调度实验

2. 实验目的

本实验的目的是通过作业或进程调度算法模拟设计,进一步加深对作业或进程调度算法的理解,通过计算平均周转时间和带权平均周转时间,进一步加深对算法的评价方法的理解。

3. 实验预备内容

(1) 掌握作业或进程调度算法。

(2) 平均周转时间和带权平均周转时间计算。

4. 实验内容

设定一组作业或进程,给定相关参数,对这组进程或作业按调度算法实施调度,输出调度次序,并计算平均周转时间和带权平均周转时间。使用的调度算法有:

(1) 先来先服务调度算法;

(2) 优先级调度算法;

（3）短作业（或进程）优先调度算法；

（4）响应比高优先调度算法。

5. 提示

1）使用的主要数据结构

（1）定义一个结构体，结构体的主要成员有序号、作业（进程）号或名称、提交时间、运行时间、优先数、进入输入井时间、开始运行时间、尚需运行时间、运行结束时间、周转时间、带权周转时间和运行次序等。

（2）利用定义的结构体，定义一个结构体数组，用来记录系统中的作业或进程。

2）算法描述

主控程序算法、数据输入算法、数据输出算法、先来先服务调度算法、响应比高优先调度算法、优先级调度算法和短作业（或进程）优先调度算法描述分别如图 10.2 至图 10.8 所示。

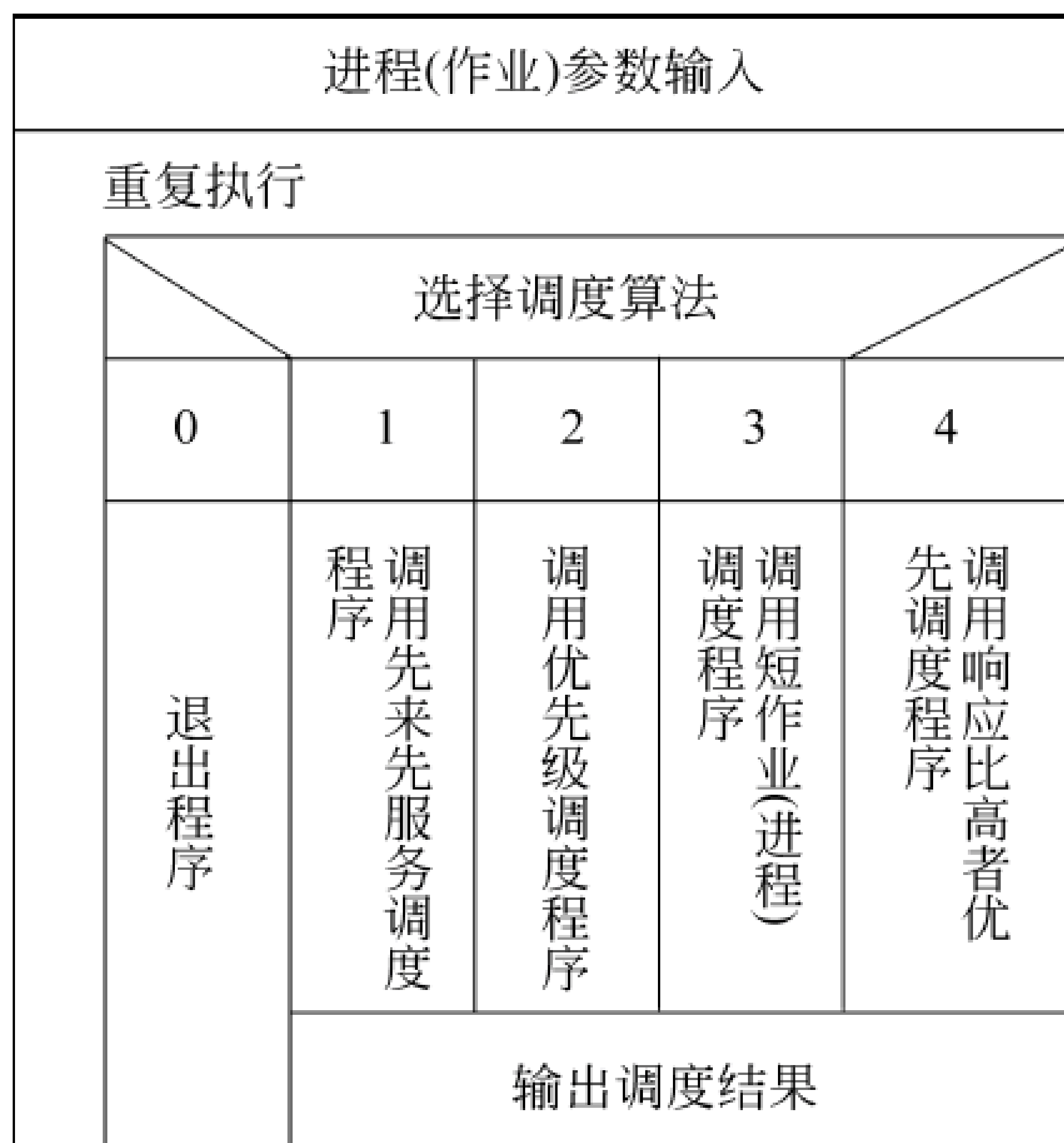


图 10.2 主控程序算法

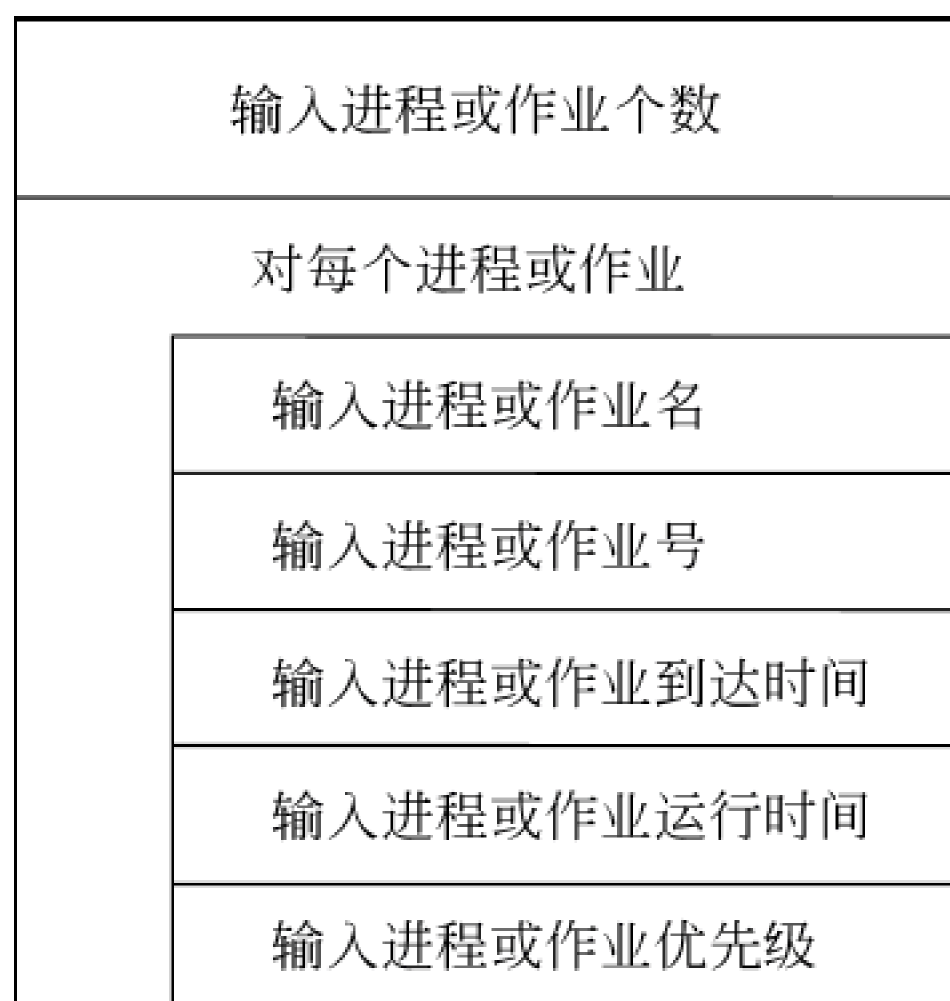


图 10.3 数据输入算法

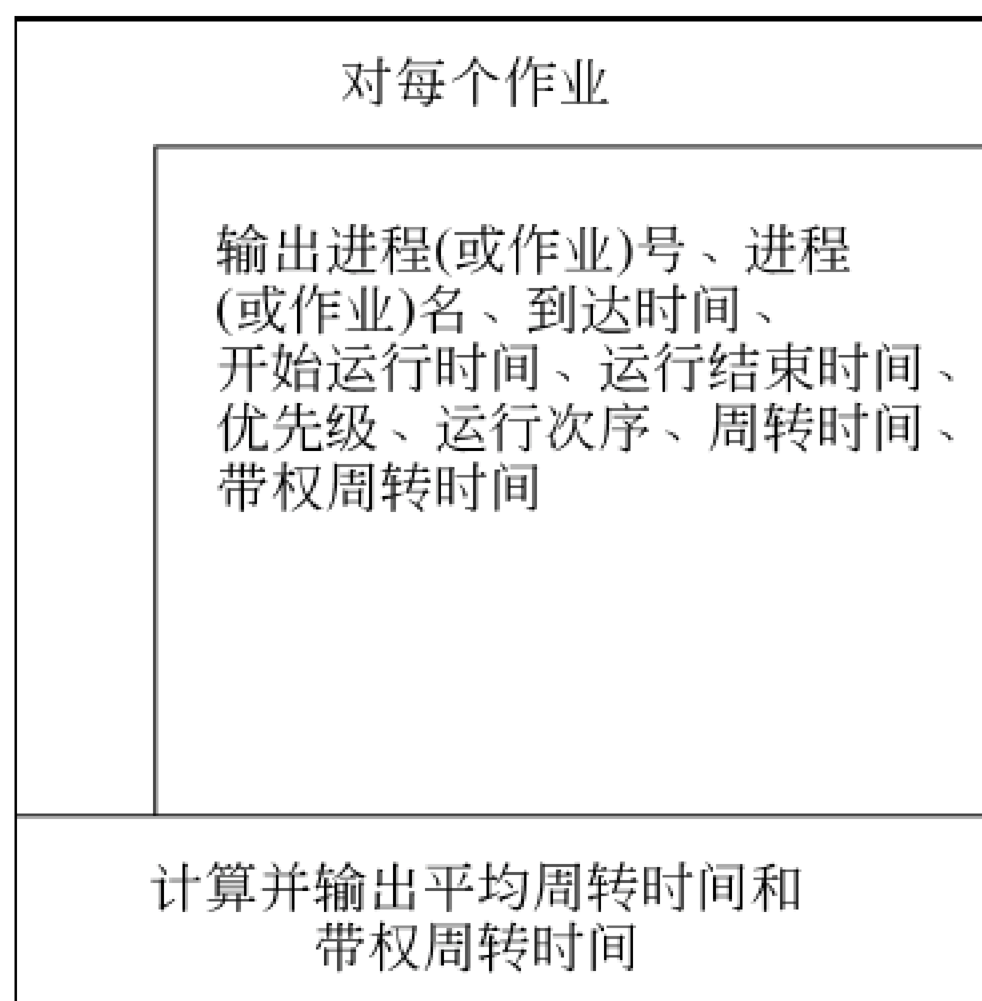


图 10.4 数据输出算法

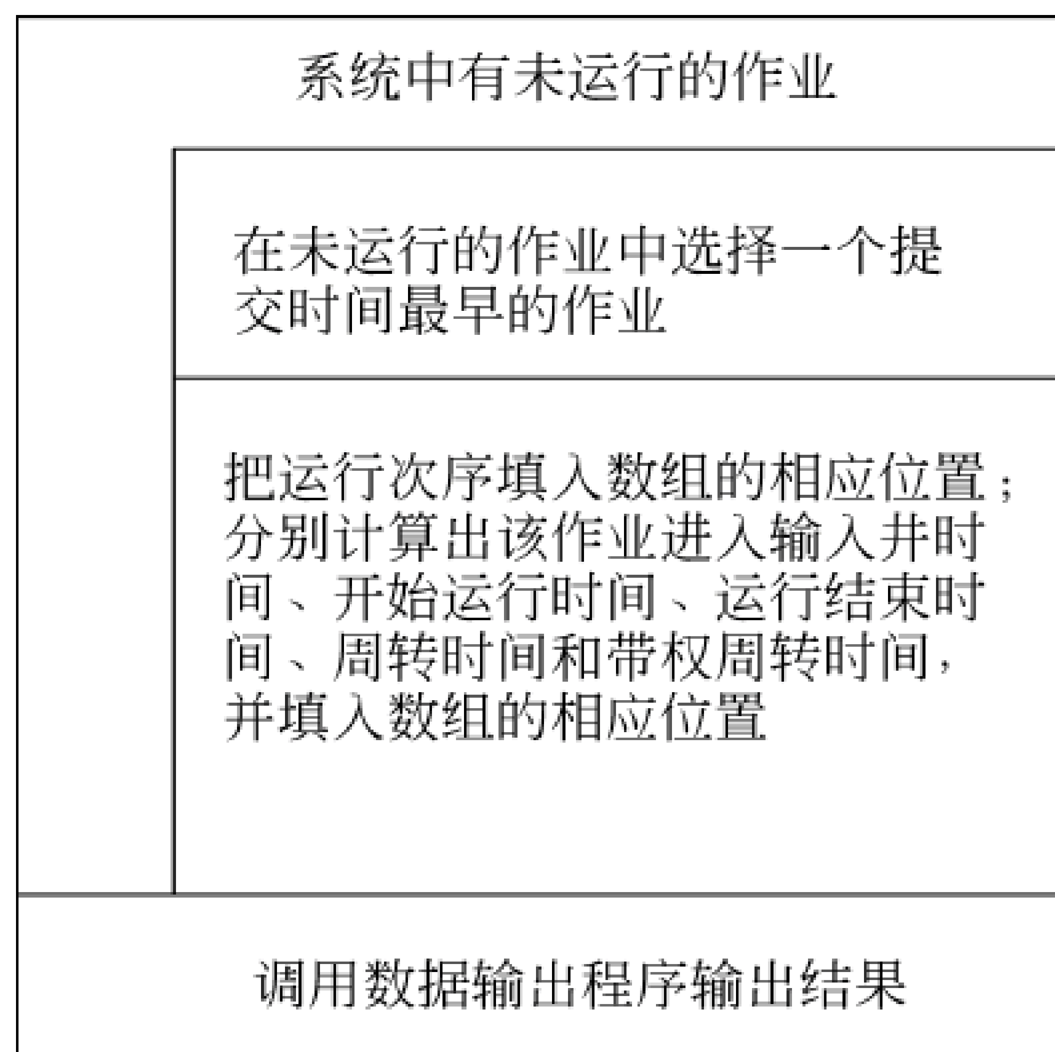


图 10.5 先来先服务调度算法

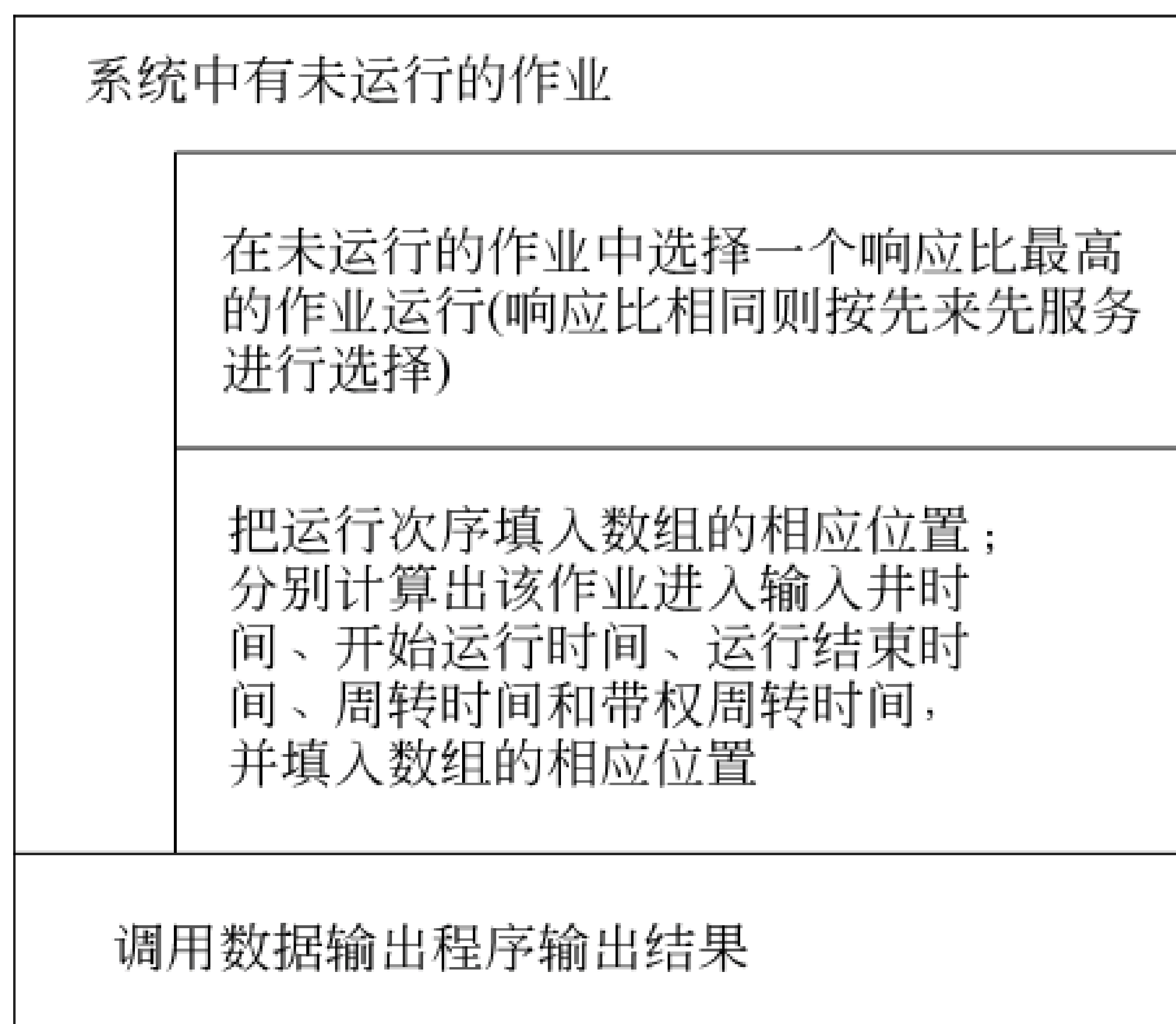
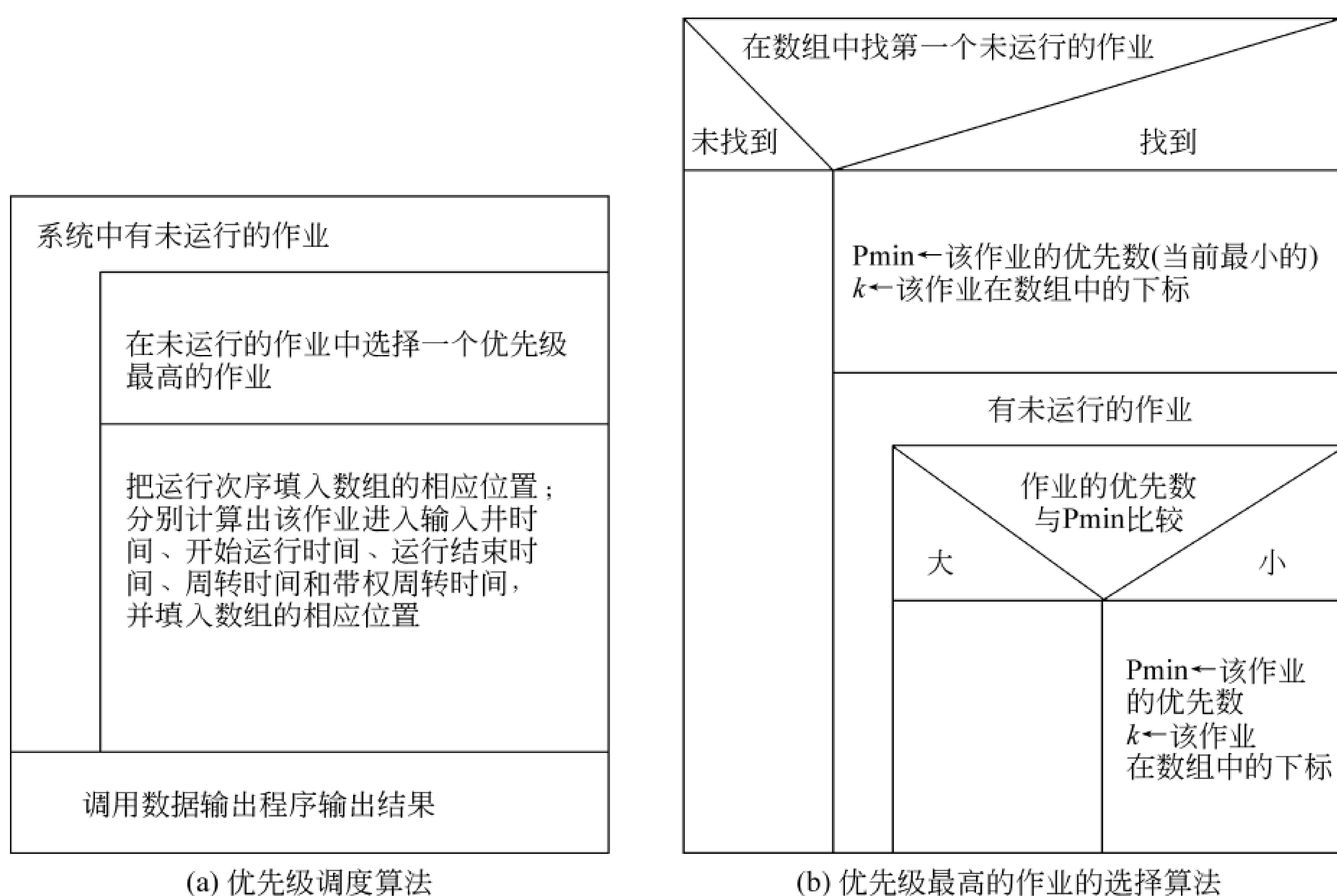
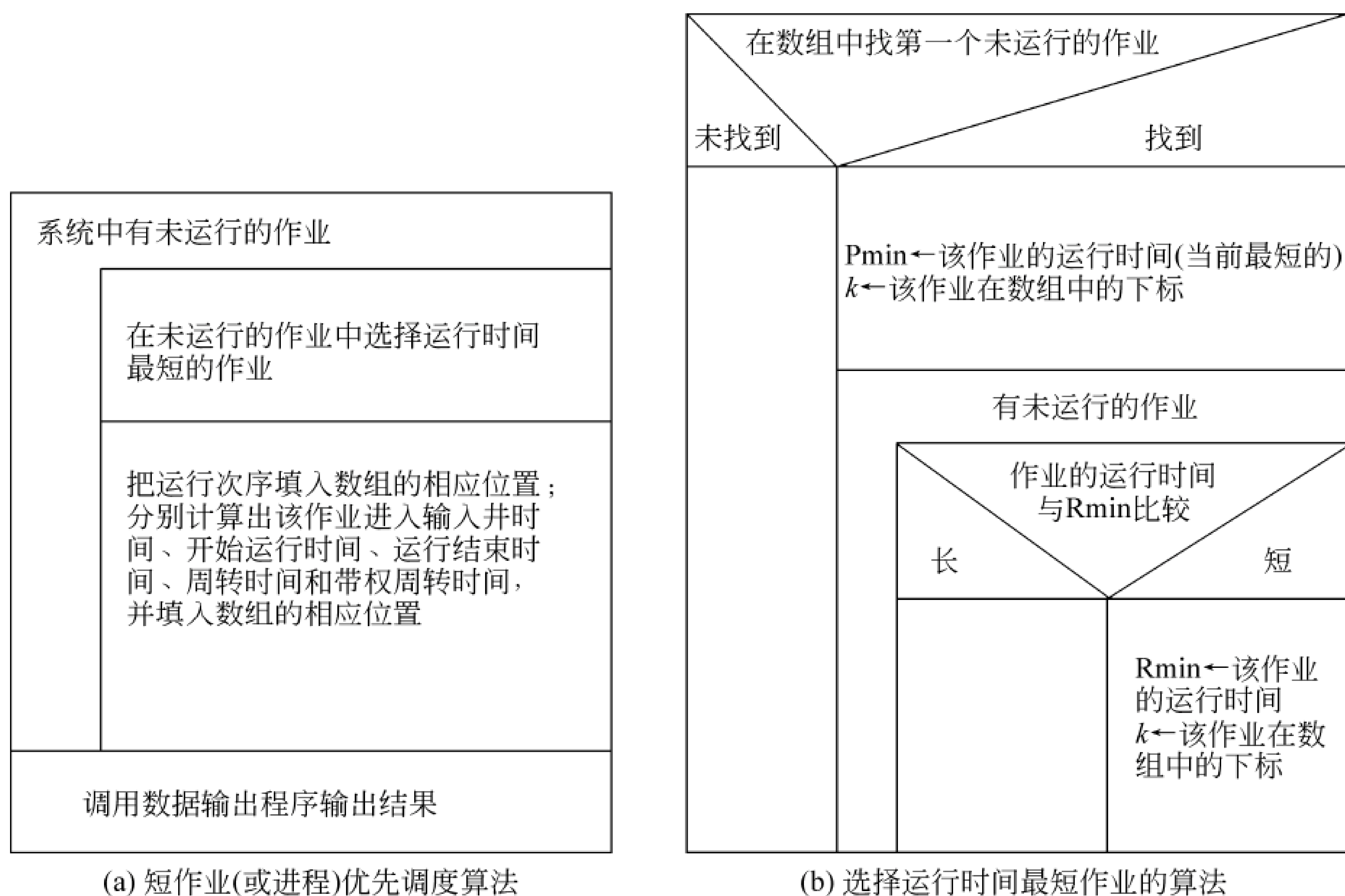


图 10.6 响应比高优先调度算法



(a) 优先级调度算法　　(b) 优先级最高的作业的选择算法

图 10.7　优先级调度算法及优先级最高的作业的选择算法



(a) 短作业(或进程)优先调度算法　　(b) 选择运行时间最短作业的算法

图 10.8　短作业(进程)优先调度算法及选择运行时间最短的作业的算法

3）实验用数据

表 10.1 给出实验用的一组参考数据。

表 10.1　实验3实验用数据

序号	进程名	进程号	到达时间	运行时间	优先级	开始运行时间	运行结束时间	周转时间	带权周转时间
1	A	1	7	2	1				
2	B	2	8	1	3				
3	C	3	8.5	2	4				
4	D	4	9	0.5	2				

10.4　动态页式存储管理实验

1. 实验题目

动态页式存储管理实验

2. 实验目的

本实验的目的是通过请求页式存储管理中的页面调度算法模拟设计，了解虚拟存储技术的特点，掌握请求页式存储管理中的页面调度算法，并会计算缺页中断率。

3. 实验预备内容

(1) 掌握请求页式存储管理中的页面调度算法。

(2) 通过一个指令序列计算缺页中断率。

4. 实验内容

设定一个指令序列，设定内存中分配的页数。模拟指令序列的执行，将指令流转换为地址流，指出该地址是否在内存，如果不在内存，输出淘汰的页和调入的页；如果在内存，输出其页号和页内地址，并计算缺页中断率。使用的页面淘汰算法为先进先出的算法。即模拟页式虚拟存储管理中硬件的地址转换和缺页中断，并用先进先出调度算法(FIFO)处理缺页中断。

5. 提示

1) 实验的进一步说明

(1) 为了调入一页而必须调出一页时，如果被选中调出的页面在执行中没有被修改过，则不必把该页重新写到磁盘上。因此在页表中可以增加是否修改过的标志，当执行“存”指令时把对应页的修改标志置成“1”表示该页修改过，为“0”表示没有修改过。页表格式如表10.2所示。

表 10.2　页表格式

页号	标志	主存块号	修改标志	在磁盘上的位置

(2) 设计一个地址转换程序来模拟硬件的地址转换和缺页中断。当访问的页在主存时则形成绝对地址，但不去模拟指令的执行，可用输出转换后的绝对地址来表示一条指令已完成。当访问的页不在主存时，则输出“第 L 页不在主存”来表示硬件产生了一次缺页中断。

(3) FIFO 页面调度算法总是先调出作业中最先进入主存的那一页,因此可以用一个数组来构成页号队列。数组中每个元素是该作业已在主存的页,假定分配给作业的主存块数为 M,且该作业开始的 M 页已调入主存,则数组可由 M 个元素组成:

$$P[0],P[1],P[2],\cdots,P[M]$$

它们的初值为

$$P[0]=0,P[1]=1,\cdots,P[M-1]=M-1$$

用指针 K 指示当要调入新页时应调出的页在数组中的位置,K 的初值为"0"。

当产生缺页中断后,操作系统总是选择 $P[K]$所指示的页面调出,然后执行

$$P[K]=\text{要调入的新页页号}$$

$$K=(K+1)\text{MOD } M$$

在实验中不必实际地启动磁盘执行调出一页和调入一页的工作,而用输出"把 J 页调出内存"和"把 L 页调入内存"来模拟一次调出和调入的过程。模拟程序的算法描述如图 10.9 所示。

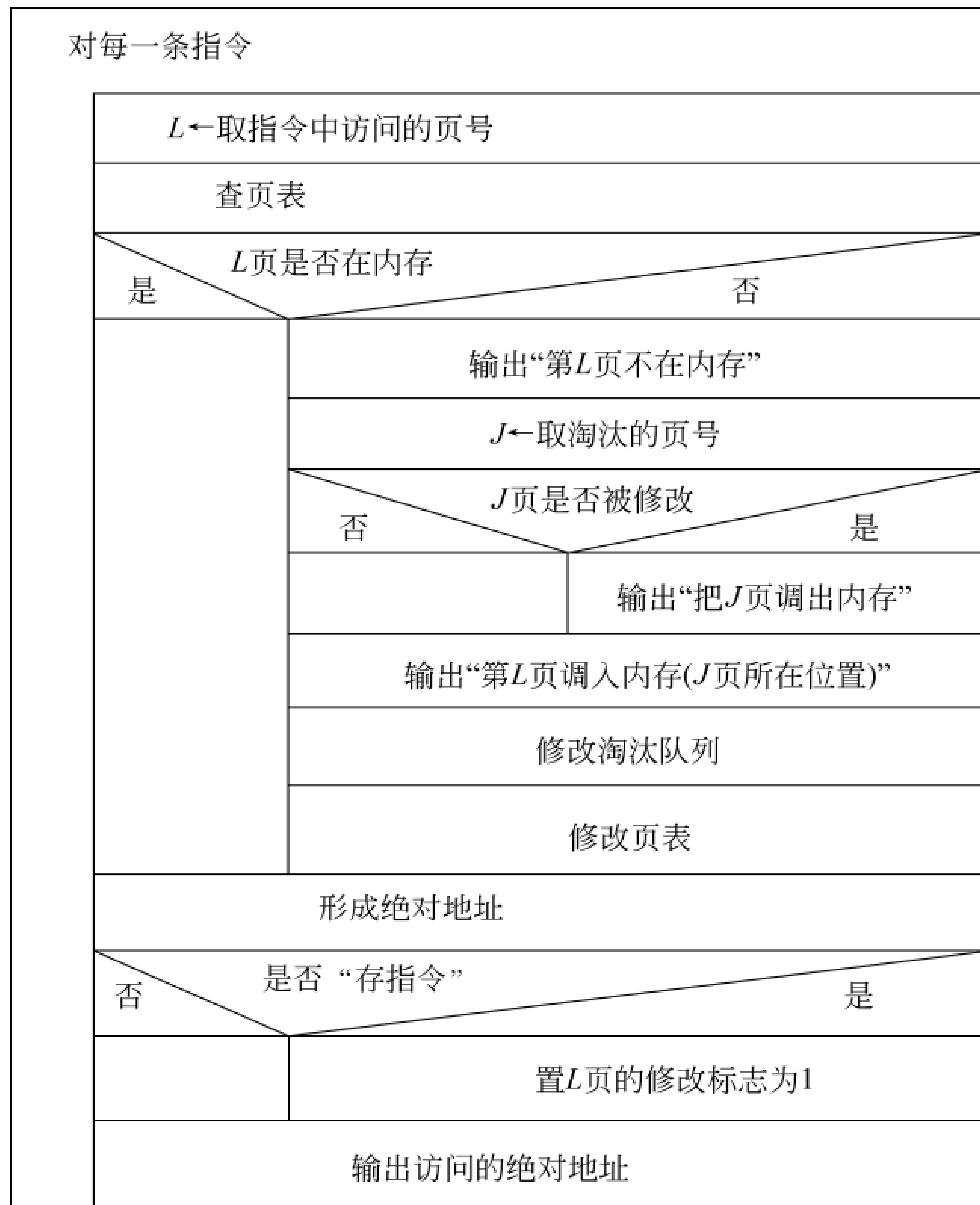


图 10.9 模拟动态页式存储管理算法

(4) 假定主存的每块长度为 1024B,现有一个共 7 页的作业,其副本已在磁盘上。系统为该作业分配了 4 个主存块,且该作业的第 0 页至第 3 页已经调入主存,其余 3 页尚未调入主存,该作业的页表见表 10.3。该作业依次执行的指令序列如表 10.4 所示。

表 10.3　页表实验数据

页　号	标　志	主存块号	修改标志	在磁盘上的位置
0	1	5	0	11
1	1	8	0	12
2	1	9	0	13
3	1	1	0	21
4	0		0	22
5	0		0	23
6	0		0	121

表 10.4　实验数据（指令执行的操作和访问的地址）

操　作	页　号	页内地址	操　作	页　号	页内地址
+	0	70	M(移)	4	53
+	1	50	+	5	23
*	2	15	S	1	37
S(存)	3	21	L	2	78
L(取)	0	56	+	4	1
—	6	40	S	6	84

2）使用的数据结构

表 10.2 给出了实验用的页表的形式。

定义一个结构体数组，用来表示页表，该结构体数组定义的具体形式见程序部分。

3）算法描述

4）实验用数据

表 10.3 给出了一组页表实验数据，表 10.4 给出了一组实验用指令序列。

10.5　文件系统实验

1. 实验题目

文件系统实验

2. 实验目的

本实验的目的是通过一个简单的多用户二级文件系统的设计，加深理解文件系统的内部功能和实现方式。

3. 实验预备内容

(1) 掌握文件系统的管理。

(2) 了解文件系统的功能。

4. 实验内容

设计一个简单的多用户二级文件系统，要求该文件系统具有以下功能：

(1) 用户登录。

(2) 创建文件。
(3) 删除文件。
(4) 打开文件。
(5) 关闭文件。
(6) 读文件。
(7) 写文件。
(8) 列文件目录。

5. 提示

1) 使用的数据结构

表 10.5 给出了一级目录(主目录 MFD)格式,表 10.6 给出了二级目录(用户文件目录 UFD)格式,表 10.7 给出了已打开文件名表(UOF)。

表 10.5 一级目录(主目录 MFD)格式

用 户 名	用户文件目录地址
(字符型(20))	(整数型)

表 10.6 二级目录(用户文件目录 UFD)

文 件 名	文件属性	记录长度	文件地址
(字符型(20))	(字符型(10))	(整数型)	(整数型)

表 10.7 已打开文件名表(UOF)

文 件 名	文 件 属 性	记 录 长 度	状态(打开/建立)	读 指 针	写 指 针
(字符型(20))	(字符型(10))	(整数型)	(整数型(1: 建立,0: 打开))	(整数型)	(整数型)

定义 3 个结构体数组,分别用来记录主目录、用户文件目录和已打开文件名,定义的具体形式见程序部分。

2) 算法描述

主程序算法、建立文件算法、打开文件算法、写文件算法、读文件算法、关闭文件算法、删除文件算法和列文件目录算法分别如图 10.10 至图 10.17 所示。

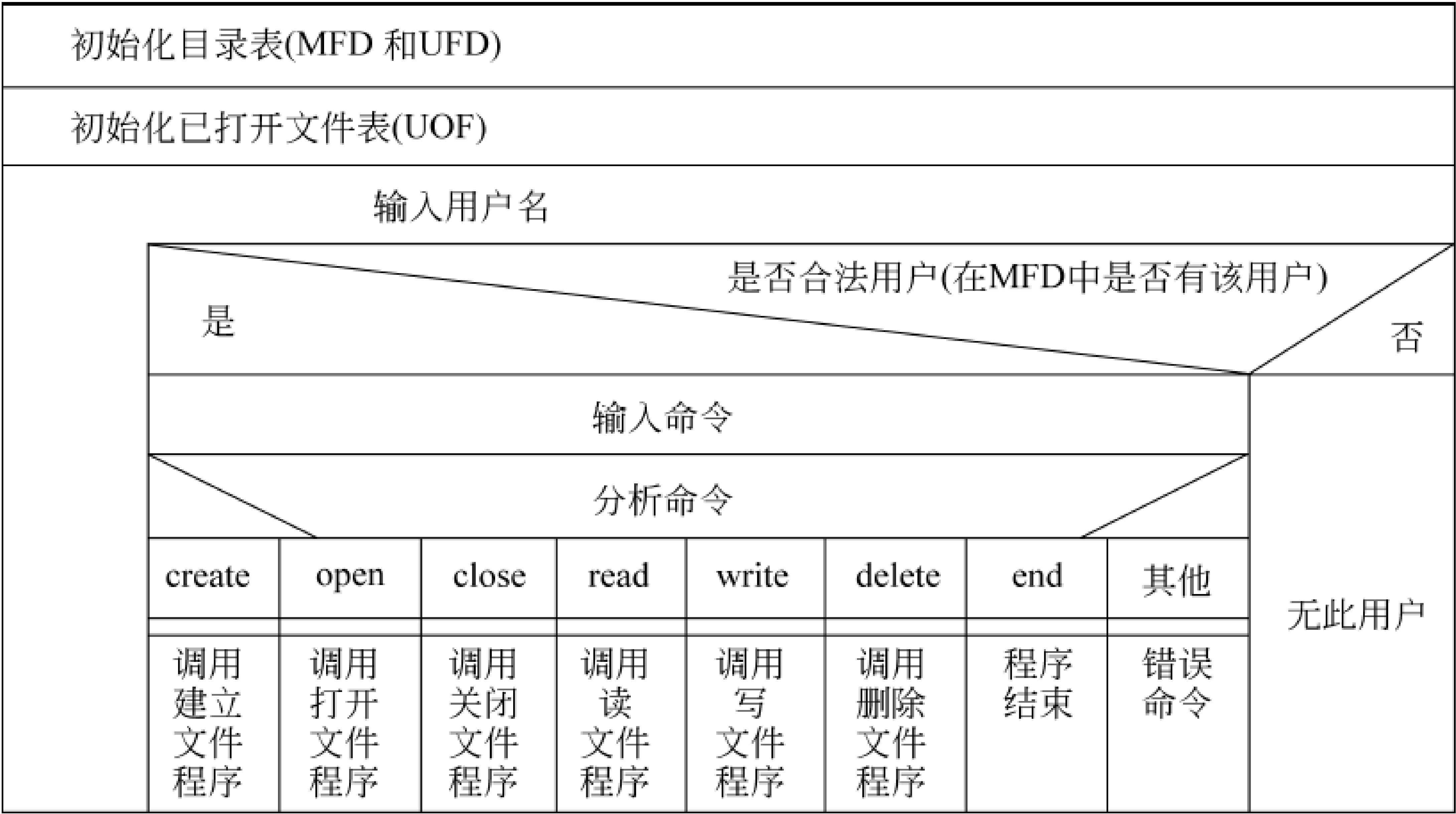


图 10.10 主程序算法描述

查该用户的UFD

UFD中有该文件

无　　有

在UFD中找空登记栏

有　　无

在UOF中找空登记栏

找一磁盘空闲块

有　　无

空闲块的块号记为i

在UFD和UOF中登记文件名、记录长度和文件属性

在UFD中的登记栏中登记文件地址为i

在UFD和UOF中的登记栏中登记状态为“建立”，写指针为i

显示“建立成功”

显示“磁盘无空闲块，不能建立”

显示“在UFD中无空登记栏，不能建立”

显示“有同名文件，不能建立”

图 10.11　建立文件算法描述

查该用户的UFD

UFD中有该文件

有　　无

查用户的UOF

UOF中有该文件

无　　有

文件属性与操作类型相符

是　　否

该文件为建立状态

是　　否

在UOF中登记文件名、记录长度和文件属性

写指针为UOF中的文件地址
读指针为UOF中的文件地址

显示“打开成功”

显示“操作不合法，不能打开”

显示“正在建立，不能打开”

显示“文件已打开”

显示“文件不存在，不能打开”

图 10.12　打开文件算法描述

查该用户的UOF

UOF中有该文件

有

无

该文件为建立状态

否

是

文件属性为“只读”

否

是

顺序修改

否

是

找出存放指定记录的块号

取出“写指针”指出的块号

修改“写指针”

把记录信息写入找到的磁盘块中(用显示磁盘块号来模拟)

显示“写文件成功”

显示“操作不合法，不能写”

把记录信息写到“写指针”指出的磁盘块中(用显示磁盘块号来模拟)

找一个空闲块记录文件连接字，修改“写指针”

显示“写文件成功”

显示“文件不存在，不能写”

图 10.13　写文件算法描述

查该用户的UOF

UOF中有该文件

有

无

从“读指针”得到当前起始地址

按读长度把记录信息读出送给用户(用显示磁盘块号来模拟)

修改“读指针”

显示“读文件成功”

显示“文件未打开，不能读”

图 10.14　读文件算法描述

查该用户的UOF

文件为“建立”状态

否

是

文件为“打开”状态

否

是

置文件结束标志

显示“文件已打开”

消除该文件在UOF中的登记栏

显示“关闭文件成功”

图 10.15　关闭文件算法描述

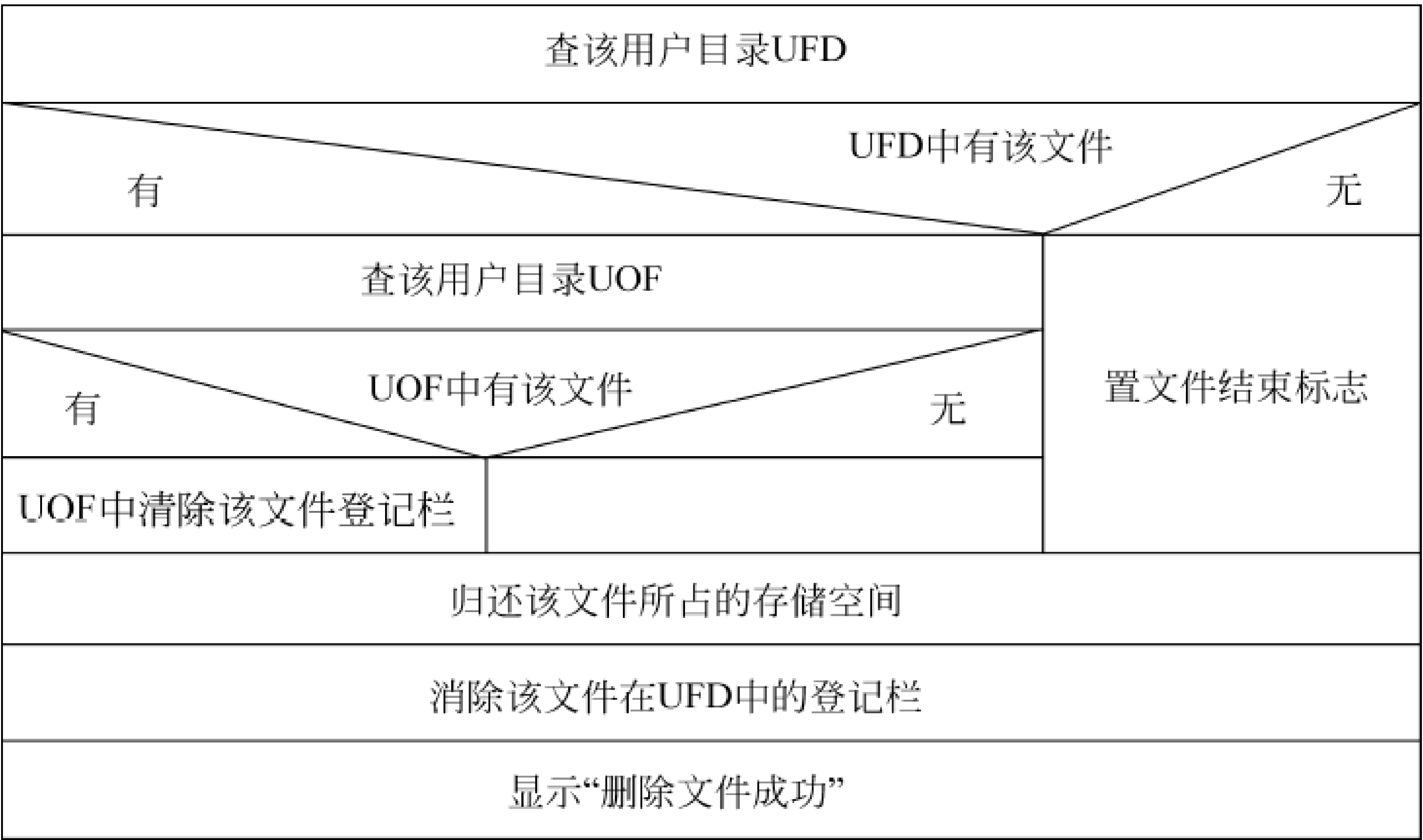


图 10.16　删除文件算法描述

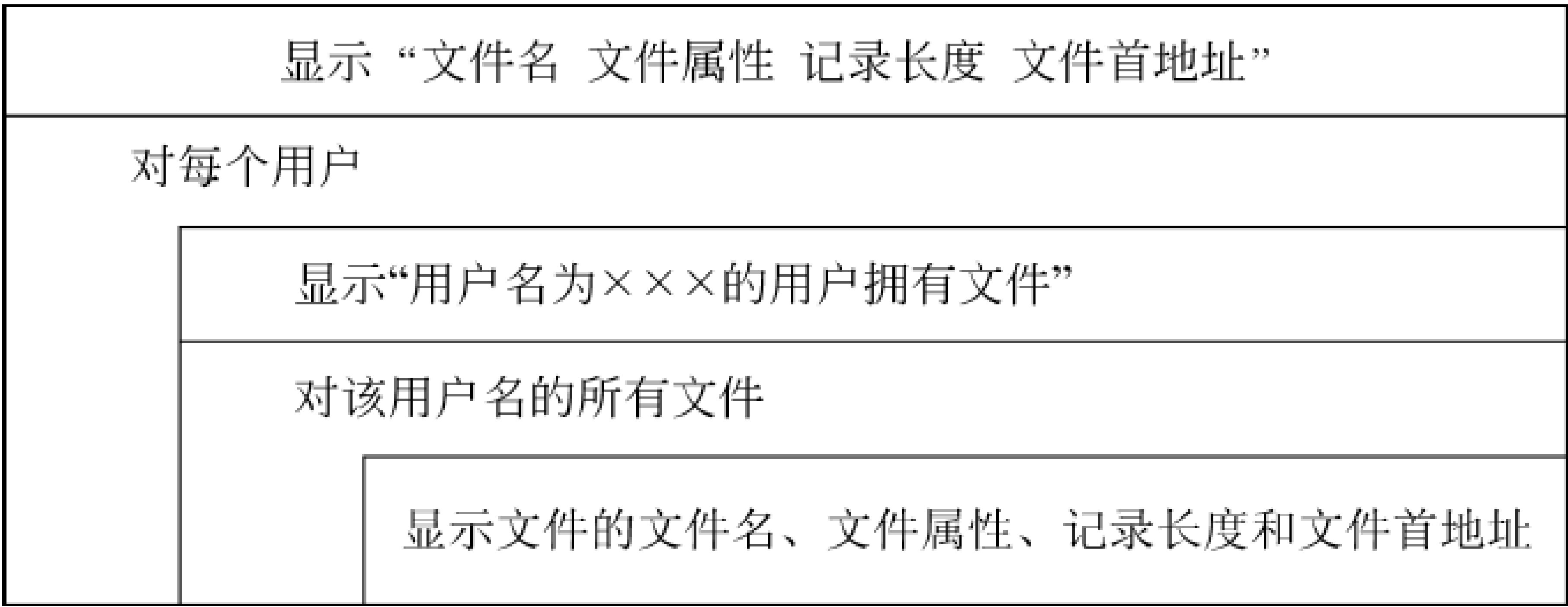


图 10.17　列文件目录算法描述

3）实验用数据

表 10.8 给出了一组主文件目录实验用数据，表 10.9 给出了一组用户文件目录实验用数据，表 10.10 给出了一组用户打开文件名实验用数据。

表 10.8　主文件目录实验用数据（最多 10 项）

uname（用户名）	uaddr（首地址）	uname（用户名）	uaddr（首地址）
user1	20	user6	220
user2	60	user7	260
user3	100	user8	300
user4	140	user9	340
user5	180	user10	380

表 10.9　用户文件目录实验用数据（每个用户最多 10 个文件）

所属用户	fname	fattr	record1	addrf
user1	u1f1		10	22
user1	u1f2		20	24

续表

所属用户	fname	fattr	record1	addrf
user1	u1f3		30	26
user2	u2f1		25	62
user2	u2f2		35	64
user2	u3f1		15	102
user2	u3f2		5	104
user5	u5f1		40	142
user5	u5f2		45	144

表 10.10 用户打开文件名表(每个用户最多 6 个)

fname	fattr	record1	fstatue	readp	writep
u1f1	r	10	1	70	
u1f3	w	30	0		10
u2f1	w	25	0		50
u3f1	w	5	1		40
u5f1	r	40	1	60	
u5f2	w	45	0		80

10.6 进程管理(同步、互斥和通信)实验

1. 实验题目

进程互斥、同步和通信实验

2. 实验目的

本实验的目的是通过建立一个简单的进程互斥、同步和通信程序加深对进程的管理和控制内容的理解。

3. 实验预备内容

了解 Linux 系统的有关函数：wait()、sleep()、lockf()和 pipe()等。

4. 实验内容

编写一个程序,利用管道实现父子进程间的通信。

(1) 创建一个管道。

(2) 父进程创建两个子进程。

(3) 利用管道实现父进程与两个子进程间的通信。

5. 提示

1) 管道通信原理

通过管道可进行父子进程间的通信。管道在物理上由文件系统的高速缓冲区(buf)构

成，一个管道 fd 有两端：一端为写端 fd[1]，另一端为读端 fd[0]。发送进程利用文件系统的系统调用 write(fd[1],buf,size)把 buf 中长度为 size 字符的消息送入管道入口 fd[1]，接收进程利用文件系统的系统调用 read(fd[0],buf,size)从管道出口 fd[0]读出长度为 size 字符的消息并送入 buf 中。管道通信原理如图 10.18 所示。利用管道实现父进程与两个子进程间的通信原理如图 10.19 所示。

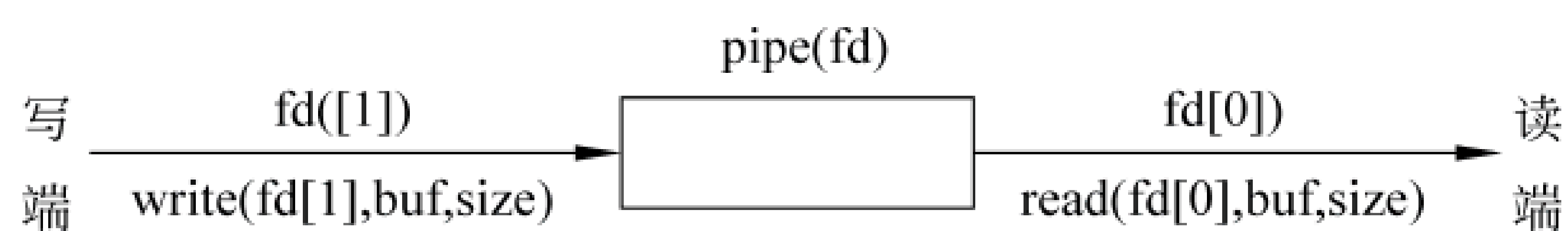


图 10.18　管道通信

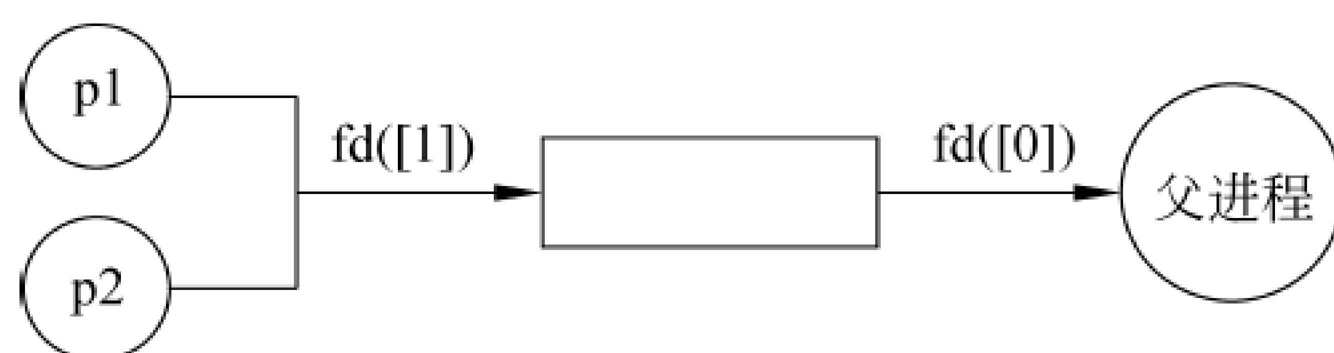


图 10.19　父进程与两个子进程 p1、p2 通信

2）程序实现

利用管道实现父进程与两个子进程间的通信用 C 语言所编程序如下：

```
#include< stdio.h>
int main()
{
    int i,r,p1,p2,fd[2];
    char buff[50],str[50];
    pipe(fd);
    while((p1=fork())==-1);
    if(p1==0)
    {
        lockf(fd[1]),1,0);
        sprintf(buff,"child process p1 is sending messages!\n");
        printf("child process p1!\n");
        write(fd[1],buff,50);
        sleep(5);
        lockf(fd[1],0,0);
        exit(0);
    }
    else
    {
        while((p2=fork())==-1);
        if(p2==0)
        {
            lockf(fd[1]),1,0);
            sprintf(buff,"child process p2 is sending messages!\n");
            printf("child process p2!\n");
```

```
            write(fd[1],buff,50);
            sleep(5);
            lockf(fd[1],0,0);
            exit(0);
        }
        wait(0);
        if ((r=read(fd[0],str,50))==-1)
            printf("can't read pipe!\n");
        else printf("%s\n",str);
            wait(0);
        if ((r=read(fd[0],str,50))==-1)
            printf("can't read pipe!\n");
        else printf("%s\n",str);
            exit(0);
    }

    return 0;
}
```

注：该实验程序在 Linux 环境中执行。

参 考 文 献

[1] Abraham Silberschatz，Peter Galvin，Greg Gagne. Applied Operating System Concepts（影印版）[M]. 北京：高等教育出版社，2003.

[2] Andrew S. Tanenbaum. Distributed Operating Systems(影印版)[M]. 北京：清华大学出版社，2000.

[3] Andrew S. Tanenbaum. Modern Operating System(影印版)[M]. 北京：机械工业出版社，2002.

[4] Andrew S. Tanenbaum. 分布式操作系统[M]. 陆丽娜，伍卫国，刘隆国，等译. 北京：电子工业出版社，1999.

[5] Barry Wilkinson，Michael Allen. Parallel Programming—Techniques and Applications Using Networked Workstations and Parallel Computers(影印版)[M]. 北京：高等教育出版社，2002.

[6] Doreen L. Galli. 分布式操作系统原理与实践[M]. 徐良贤，等译. 北京：机械工业出版社，2003.

[7] Glen Bruce，Rob Dempsey. 分布式计算的安全原理[M]. 李如豹，等译. 北京：机械工业出版社，2003.

[8] Gary Nutt. Linux 操作系统内核实习[M]. 潘登，冯锐，陆丽娜，等译. 北京：机械工业出版社，2002.

[9] William Stallings. Operating Systems—Internals and Design principles[M]. 3rd. ed. 北京：清华大学出版社，2002.

[10] William Stallings. 操作系统——精髓与设计原理[M]. 5 版. 陈渝，译. 北京：电子工业出版社，2007.

[11] 安淑芝，刘光然，杨虹，等. 操作系统原理与应用[M]. 北京：北京希望电子出版社，2002.

[12] 傅麒麟，徐勇. 现代计算机体系结构教程[M]. 北京：北京希望电子出版社，2002.

[13] 龚丽敏，刘勇，沈熙，等. 基于 Linux 的安全操作系统开发过程与质量控制的研究[J]. 计算机科学，2002，29(4)：21～23.

[14] 蒋苹，胡华平，王奕. 计算机信息系统安全体系设计[J]. 计算机工程与科学，2003，25(1)：38～41.

[15] 何炎祥. 分布式操作系统. 北京：高等教育出版社，2005.

[16] 李春葆，曾平，曾慧. 计算机操作系统联考辅导教程(2011 版)[M]. 北京：清华大学出版社，2010.

[17] 李春葆，曾平，曾慧. 计算机操作系统联考辅导教程(2012 版)[M]. 北京：清华大学出版社，2011.

[18] 梁洪亮，孙玉芳. 信息安全评价准则研究综述与探讨[J]. 计算机科学，2002，29(9)：16～20.

[19] 林建民. 嵌入式操作系统技术发展趋势[J]. 计算机工程，2001，27(10)：1～4.

[20] 陆丽娜，柯丽芳. 操作系统——重点难点及典型题精解[M]. 西安：西安交通大学出版社，2002.

[21] 罗宇，邹鹏，吴刚，等. 操作系统[M]. 北京：电子工业出版社，2003.

[22] 马华东. 多媒体技术原理与应用[M]. 北京：清华大学出版社，2003.

[23] 马季兰，彭新光. Linux 操作系统[M]. 北京：电子工业出版社，2002.

[24] 牛峰，胡昌振. 内核信息获取的通信方式[J]. 计算机工程，2003. 29(8)：114～115.

[25] 卿斯汉，刘文清，刘海峰. 操作系统安全导论[M]. 北京：科学出版社，2003.

[26] 石文昌，孙玉芳. 安全操作系统研究的发展(上)[J]. 计算机科学，2002，29(6)：5～12.

[27] 石文昌，孙玉芳. 安全操作系统研究的发展(下)[J]. 计算机科学，2002，29(7)：9～12.

[28] 史志才，毛玉萃. 操作系统原理——Linux 技术实现[M]. 北京：高等教育出版社，2006.

[29] 汤小丹，梁红兵，哲凤屏，汤子瀛. 计算机操作系统[M]. 3 版. 西安：西安电子科技大学出版社，2009.

[30] 吴鹤龄，崔林. ACM 图灵奖——计算机发展史的缩影[M]. 北京：高等教育出版社，2002.

[31] 夏靖波，王航，陈雅蓉. 嵌入式系统原理与开发[M]. 西安：西安电子科技大学出版社，2006.

[32] 徐千洋. Linux C 函数库详解词典[M]. 北京：机械工业出版社，2008.

[33] 张红光，李福才，等. UNIX 操作系统教程[M]. 北京：机械工业出版社，2003.

[34] 张尧学，史美林．计算机操作系统教程[M]．北京：清华大学出版社，2000.

[35] 张尧学．计算机操作系统教程（第 3 版）习题解答与实验指导[M]．北京：清华大学出版社，2000.

[36] 张思民．嵌入式系统设计与应用[M]．北京：清华大学出版社，2008.

[37] 钟小玲，袁宏春．Linux 的进程调度[J]．计算机应用，2002．22(1)：42～22.

[38] 钟玉琢，等．流媒体和视频服务器[M]．北京：清华大学出版社，2003.

[39] 周伟，尹青，王清贤．计算机安全中的经典模型[J]．计算机科学，2004，31(3)：195～200.